Software implemented fault tolerance for microprocessor controllers: fault tolerance for microprocessor controllers

Guy A.S. Wingate

SOFTWARE IMPLEMENTED FAULT TOLERANCE FOR MICROPROCESSOR CONTROLLERS

by

Guy A.S. Wingate, B.Sc.(Hons), M.Sc.

A Thesis submitted in fulfilment
of the requirement for the degree of
Doctor of Philosophy

Engineering Science

The University of Durham
1992

DECLARATION

None of the work contained within this thesis has previously been submitted for a degree at this or any other university. The work contained in this thesis is not part of a joint research project.

SOFTWARE IMPLEMENTED FAULT TOLERANCE FOR MICROPROCESSOR CONTROLLERS

Guy A.S. Wingate, B.Sc.(Hons), M.Sc.

ABSTRACT

It is generally accepted that transient faults are a major cause of failure in microprocessor systems. Industrial controllers with embedded microprocessors are particularly at risk from this type of failure because their working environments are prone to transient disturbances which can generate transient faults.

In order to improve the reliability of processor systems for industrial applications within a limited budget, fault tolerant techniques for uniprocessors are implemented. These techniques aim to identify characteristics of processor operation which are attributed to erroneous behaviour. Once detection is achieved, a programme of restoration activity can be initiated.

This thesis initially develops a previous model of erroneous microprocessor behaviour from which characteristics particular to mal-operation are identified. A new technique is proposed, based on software implemented fault tolerance which, by recognizing a particular behavioural characteristic, facilitates the self-detection of erroneous execution. The technique involves inserting detection mechanisms into the target software. This can be quite a complex process and so a prototype software tool called Post-programming Automated Recovery UTility (PARUT) is developed to automate the technique's application. The utility can be used to apply the proposed behavioural fault tolerant technique for a selection of target processors. Fault injection and emulation experiments assess the effectiveness of the proposed fault tolerant technique for three application programs implemented on an 8, 16, and 32-bit processors respectively. The modified application programs are shown to have an improved detection capability and hence reliability when the proposed fault tolerant technique is applied. General assessment of the technique cannot be made, however, because its effectiveness is application specific.

The thesis concludes by considering methods of generating non-hazardous application programs at the compilation stage, and design features for incorporation into the architecture of a microprocessor which inherently reduce the hazard, and increase the detection capability of the target software. Particular suggestions are made to add a 'PARUT' phase to the translation process, and to orientate microprocessor design towards the instruction opcode map.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor Dr. Clive Preece for his tremendous support and encouragement throughout the duration of the research project. I am also grateful to British Gas plc. for funding the research; in particular I would like to thank Dr. Ken Jenkins and colleagues at the Engineering Research Station, Killingworth. In addition, I owe thanks to: Prof. Ed Czeck (Computer Science, Carnegie-Mellon University, USA) for his comments on the work contained within Chapter 5 and 7; Mr. Alan Timothy (Microprocessor Centre, University of Durham) for comments on the design and implementation of an Advanced Micro Device Am29000 processor system C compiler and its implications on the work presented in Chapter 8; Prof. Dan Siewiorek (Carnegie-Mellon University, USA), Dr. Janusz Sosnowski (Warsaw Technical University, Poland), and Prof. Hermann Kopetz (Vienna Technical University, Austria) for the provision of details of their related research; and finally, Mr. Jim Roper (Computer Science, University of Durham) and Dr. Per Nylen (University of Stockholm, Sweden) for access to an Intel 80386 processor and Motorola 68(7)05 processor emulator respectively.

My time at Durham has not been all work! I would like to thank my friends over the years for making my stay so enjoyable: especially the 'tea room boys' – Sabah, Lee, Ken, Jean, and Norman; Hatfield College Middle Common Room; and the university squash and badminton teams for helping me release tension. Thanks also to Grandma for constant provision of orange cake.

Whilst writing this thesis I have phoned home many times to report "It's finished !". The natural reply soon became "Except for ... ?". Lastly then, I would like to thank my parents and my fianc'ee Sarah for their patient support and to assure them that I really have finished.

(Submitted November 1990)

The examination of this thesis was delayed by illness. I would like to thank the staff of Dryburn Hospital (County Durham), and R.N.H. Haslar (Gosport, Hants) for their care in the intervening period.

(Examined April 1992)

"Bloody instructions, which, being learned, return to plague the inventor."

'Macbeth': Act 1, Scene 7, Lines 8-10

by William Shakespeare.

Chapter 1

RELIABILITY AND MICROPROCESSOR-BASED CONTROLLERS

Chapter 2

TEMPORARY FAULTS: GENERATION, IMPLICATION, & DETECTION

Chapter 3

MODELLING ERRONEOUS MICROPROCESSOR BEHAVIOUR

Chapter 4

EVALUATING MICROPROCESSOR BEHAVIOUR

Chapter 5

DETECTING ERRONEOUS MICROPROCESSOR BEHAVIOUR

Chapter 6

POST-PROGRAMMING, AUTOMATED, RECOVERY UTILITY (PARUT)

Chapter 7

ASSESSING FAULT TOLERANCE

Chapter 8

GENERATING NON-HAZARDOUS SOFTWARE

Chapter 9

MICROPROCESSOR DESIGN FOR FAULT TOLERANCE

Chapter 10

CONCLUSION

Bibliography & References

Appendix A

INSTRUCTION SET PARAMETERS

Appendix B

THE DESIGN OF AN ACCESS GUARDIAN

Appendix C

PARUT AND OTHER RELATED CODE LISTINGS

Appendix D

EXAMPLE PROGRAMS

Appendix E

PUBLICATIONS

FIGURES

TABLES

LIST OF ABBREVIATIONS

ADB	UNIX 'A DeBugger' Facility
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Device
BG-Rig	British Gas Rig (experimental)
BIC	Bus Interface Circuitry
Cdf	Cumulative Density Function
CISC	Complex Instruction Set Computer
CLASSIC	Custom Logic Analysis Simulation System for Integrated Circuits
Cm*	Carnegie-Mellon Multi-processor
CMOS	Complementary Metal-Oxide Semiconductor
CMUA	Carnegie-Mellon University 'A' File System
CMU-AFS	Carnegie-Mellon University Andrew File System
C.vmp	Carnegie-Mellon Voting Multi-processor
DIS	UNIX 'DIS-assembler' Facility
ECL	Emitter-Coupled Logic
EMI	Electro-Magnetic Interference
ESD	Electro-Static Discharge
FMECA	Failure-Mode, Effects and Criticality Analysis
FORTRAN	'Formula Translation' Programming Language
FTA	Fault Tree Analysis
FTMP	Fault Tolerant Multi-Processor
IBM	International Business Machines
IC	Integrated Circuit
IEE	Institution of Electrical Engineers
IEEE	Institute of Electrical and Electronics Engineers
IEJ	Initial Erroneous Jump
IPQ	Instruction Prefetch Queue
HDL	Hardware description Language
JPI	Jensen and Partners International
LP	Low Pressure
MC	Motorola Corporation
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
NASA	National Aeronautics Space Administration
NMOS	N-Type Metal-Oxide Semiconductor

PARUT	Post-programming Automated Recovery UTility
PC	Program Counter
PCU	Program Control Unit
pdf	Probability Density Function
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RFI	Radio-Frequency Interference
ROM	Read Only Memory
SEJ	Subsequent Erroneous Jump
SLAC	Stanford Linear Accelerator Centre
SRFF	Set-Reset Flip Flop
TMR	Triple Modular Redundancy
TTL	Transistor-Transistor Logic
VLSI	Very Large Scale Integration

LIST OF SYMBOLS

A	Set of Faults Detected by Access Guardian
A_d	Physical Address in the Address Space
A_v	Availability
C_n	Ripple Counter Base
$D(k)$	Probability of Detecting Erroneous Execution
$D_{IEJ}(k)$	Probability of Detecting Erroneous Execution Following an IEJ
$D_{SEJ}(k)$	Probability of Detecting Erroneous Execution Following an SEJ
E	Error Detection Coverage
E_f	Event Leading to Failure
E_r	Event Leading to Recovery
E_s	Set of Events Disrupting Processor Execution
$f(t)$	Probability Density Function
F	Set of Injected Faults
$F(t)$	Cumulative Density Function
$H(l,j)$	HIT Function: Probability of an SEJ Target in Used Area
i	Number of Defined Opcodes within Instruction Set
I	Mean Period of Linear Erroneous Execution, (No. Instructions)
I_f	Microprocessor Instruction Frequency, (No. Instructions/s)
j	Jump Type Instruction
J	Set of Jump Type Instructions
J'	Return or Unspecified Jump Outcome
k	Instruction Index During Linear Erroneous Execution
l	Jump Type Instruction Location in Used Area
L	Set of Address Space Locations
L_d	Error Detection Latency (No. Instructions)
L_s	Stalling Latency (No. Instructions)
m	Integer
M	Set of Faults Detected by Hardware
n	Number of Bits in Opcode Format
Ne_J	Effective Number of 'Jump' Outcome Instructions
Ne_{RN}	Effective Number of 'Return' Outcome Instructions
Ne_{RT}	Effective Number of 'Restart' Outcome Instructions
Ne_{SW}	Effective Number of 'Stop/Wait' Outcome Instructions
Ne_{UJ}	Effective Number of 'Unspecified Jump' Outcome Instructions
N_L	Number of Locations Open to Instruction Interpretation

N_R	Number of Instructions in Recovery Routine
P	Set of Faults Detected by Software
$P_c(t)$	Probability of Controlled State
$P_{IEJ}(UsedArea)$	Probability of IEJ Target in Used Area
$P_{IEJ}(UnusedArea)$	Probability of IEJ Target in Unused Area
P_J	Probability of a 'Jump' Instruction Outcome
$P_J(k)$	Functional Probability of a 'Jump' Outcome
P_J	Probability of a 'Jump' Instruction Outcome
$P_{J'}(k)$	Functional Probability of a 'J'' Outcome
$P_{J'}^y(k)$	Functional Probability of a 'J'' Outcome Following an IEJ or SEJ
$P_L^y(k)$	Functional Probability of Linear Erroneous Execution
P_{NJ}	Probability of a 'Non-Jump' Instruction Outcome
$P_{NJ}(k)$	Functional Probability of a 'Non-Jump' Instruction Outcome
$P_{RN}(k)$	Functional Probability of a 'Return' Instruction Outcome
$P_{RT}(k)$	Functional Probability of a 'Restart' Instruction outcome
$P_{SEJ}(UsedArea)$	Probability of SEJ Target in Used Area
$P_{SEJ}(UnusedArea)$	Probability of SEJ Target in Unused Area
$P_{SW}(k)$	Functional Probability of a 'Stop/Wait' Instruction Outcome
$P_{UJ}(k)$	Functional Probability of a 'Unspecified Jump' Instruction Outcome
$P_u(t)$	Probability of Uncontrolled State
$P_x(k)$	Functional Probability of a 'x' Outcome
$P_x^{IEJ}(k)$	$P_x(k)$ following an IEJ
$P_x^{SEJ}(k)$	$P_x(k)$ following an SEJ
$Pr\{\}$	Probability Function
q	Event Rate, (per hour)
R	Set of Faults Generating Re-synchronization
$R()$	Reliability Function
$R^F()$	Reliability Signature Monitoring Hardware
$R_s()$	Reliability of Microprocessor Employing Signature Monitoring
RN	Return Outcome
RT	Restart Outcome
s	Number of Bytes in Memory Fetch
$S(k)$	Probability of Erroneous Execution Stall
$S_{IEJ}(k)$	Probability of Erroneous Execution Stall following an IEJ
$S_{SEJ}(k)$	Probability of Erroneous Execution Stall following an SEJ
SW	Stop/Wait Outcome
t	Time, (s)
T	Set of Locations Addressed by Jump Type Instruction
UJ	Unspecified Jump Outcome
x	Member of the Set $\{RN, RT, SW, UJ\}$

y	Member of the Set $\{IEJ, SEJ\}$
$Z(t)$	Hazard Rate, (per hour)
β	Probability Hardware Exception: 'Restart' Outcome
γ	Probability Software Exception: 'Restart' Outcome
λ	Constant Failure Rate, (per hour)
$\lambda(t)$	Failure Rate, (per hour)
Ψ	Recursive Element of Function $D(k)$ and $S(k)$

Chapter 1

RELIABILITY AND MICROPROCESSOR-BASED CONTROLLERS

CHAPTER ONE

RELIABILITY AND MICROPROCESSOR-BASED CONTROLLERS

1.1. Introduction

The continuing technological evolution of microprocessor design has resulted in a large number of commercially available devices with a wide variety of characteristics. Many microprocessors are embedded within control systems to automatically operate, monitor, or control a physical process. Applications range from relatively simple control of domestic appliances such as toasters or washing machines, to the complex control of industrial plant such as power stations or chemical works. An important feature of all these control systems is their reliability, that is, the probability that the system will perform its function under stated environmental conditions, without malfunction, over a specified period of time or operational duration (adapted from [Bennetts, 1979]).

British Gas use microprocessor systems to manage individual governors controlling the transmission of gas to industrial and public customers. Such microprocessor controllers require high reliability because of the proximity of their installation, with the hazard of gas, to the general public.

The United Kingdom gas distribution system involves the transmission of gas at high- and medium- low pressures between the off-shore gas fields, local consumer districts, and end-customers respectively. The low pressure (LP) system dates back to the production of town gas in the 19th century and serves local districts and individual users. The LP network has traditionally been controlled by independent pneumatic governors. Governors are devices which control, through a valve action, the gas pressure in a pipeline. The governors implement a clocking mechanism which alters the gas pressure depending on predicted daily fluctuations in demand. Gas pressures that fall below a critical level allow air to enter the gas pipework and can produce potentially explosive mixtures. To prevent this situation arising the governor system implements a shut-down operation, called 'slam-open', when the

gas pressure falls at an excessive rate or when the gas pressure falls beneath the statutory safety limit of 12.3 mbar. The slam-open operation involves completely opening the governor valve so that adjacent normal and hazardous gas pressures across the valve are equalized. Slam-open operations may cascade through several consecutive governor systems before the mean pressure along a section of pipeline is acceptable.

In order to increase management efficiency of the LP network it is necessary to improve control of the gas distribution. The small value of gas handled by each governor system in the gas network means that a low-cost upgrade is required. To this end, microprocessors have been embedded within the governor control systems [Clark et al, 1987] to facilitate more effective and integrated control of the LP gas distribution system [Wynne et al, 1988]. The distribution of gas can now be managed in an efficient and interactive manner. Seasonal and diurnal load variations can be monitored and appropriate responsive action taken automatically in a real-time environment. Stringent safety regulations of the gas industry require the back-up of electronic systems failure by traditional pneumatic slam-open operation.

British Gas predict failures of the microprocessor assisted governors to occur once every 10 years, a failure rate approximately one thousandth of the original pneumatic governor systems [Clark et al, 1987]. The slam-open back-up ensures that microprocessor controller failures are not catastrophic. Nevertheless, loss of the governor management function incurs a financial penalty; reduced gas distribution efficiency, and repair of the controller which may reside at a remote site. An overall improvement in the reliability of the microprocessor controller is required. British Gas are particularly interested in the effect of software, 'which may become highly unpredictable' under the influence of hardware faults, on system reliability [Clark et al, 1987].

This thesis presents techniques for enhancing the operational reliability of microprocessor-based controllers without adversely increasing their cost. These techniques are applicable to any microprocessor system, including those used by British Gas to manage governors.

1.2. Microprocessor-Based Control Systems

Many control systems have their designs based on complex logic circuits incorporating flip-flops, analogue-to-digital converters, shift-registers, and other logic gate structures. In these cases it is often convenient to incorporate, or replace, such circuitry with a dedicated microprocessor and its support chips. The control system behaviour can now to a large extent be governed by the software stored on a microprocessor memory chip. Software can be maintained without physically altering the system hardware. Such flexibility can be valuable as in a recently reported incident when a car manufacturer discovered a design error in a fuel injection system [IEEE, 1989]. Replacing the system with an alternative was extremely expensive. However, replacement was not necessary: the system was controlled by a microprocessor which was re-programmed to compensate for, and effectively mask, the design error. The problem was rectified at little cost. In such instances the maintenance engineers must be careful not to introduce new errors (the 'Software Death Cycle', [Rigby & Norris, 1990]).

In recent years digital techniques have become so powerful that tasks well suited to analogue systems are often partially or totally controlled by digital systems. For example, a temperature meter based on a thermocouple or thermistor might incorporate a microprocessor and memory in order to improve accuracy by compensating for the instrument's departure from perfect linearity.

Although ever more powerful microprocessors are being developed, most controllers do not require further advanced processing capabilities. A recent Japanese survey [Fujimura, 1989] reported that approximately 80% of microprocessor-based controllers incorporated either an 8 or 16-bit machine. Despite the commercial availability of 32-bit processors for over five years, they were only used in approximately 11% of the controllers. The remaining 9% of controllers had embedded 4-bit microprocessors. Obviously the 4-bit microprocessor-based controllers had a very simple function.

1.3. Faults, Errors, and Failures in Electronic Systems

A system failure is said to occur when the behaviour of the system first deviates from that required by the specification of the system (as defined by [Anderson & Lee, 1982]). System failures are caused by the external exposure of a defective internal state. Deficiencies in the internal state of a system, referred to as 'errors', can exist without the generation of a failure.

A system consists of a set of components (or sub-systems) which interact under the control of a design. Errors originate from the activation of defective system components. Defective components are referred to as 'faults'.

A fault in a digital electronic system is characterized by its nature, extent, and duration [Avizienis, 1976]. The nature of the fault can be classified as either logical or non-logical. A logical fault causes the logic value at a point in the digital circuit to become opposite to the specified value. Non-logical faults include the remaining faults such as a malfunction of the clock signal. The extent of a fault specifies whether the effect of the fault is localized or distributed in the the digital system. Finally, the duration of a fault refers to whether the fault is permanent or temporary.

McCluskey & Wakerly [1981] distinguish between two classes of temporary fault, transient and intermittent. Transient faults are non-recurring temporary faults which are caused by environmental influences. They are not repairable because there is no physical damage to the hardware. Intermittent faults are recurring temporary faults caused by deteriorating or ageing hardware. Intermittent faults may eventually become permanent and can be repaired.

1.4. Engineering Reliability Through Design

Reliability can be engineered in a digital system by implementing a disciplined design process. The reliability of a microprocessor-based system depends on its hardware and software design, deficiencies in either are expensive to correct. It is therefore prudent, when developing reliable systems, for the design to be fault-free or fault-tolerant.

1.4.1. Reliable Hardware

Poor specification, design, and manufacture can individually or collectively introduce faults within digital electronic systems. Specifications are normally written in natural language which makes their integrity extremely difficult to check. Specifications can be written using Formal Methods, enabling designs to be proven to comply with their specification, but this technique does not ensure that the specification itself is defect-free [Cullyer, 1988].

The integrity of manufacture can be validated using 'black-box' tests. Digital systems, however, can be complex and comprehensive black-box testing extremely expensive. Intel only test 98% of the logic nodes for faults in each manufactured 80486 microprocessor (even though untested nodes may be faulty) because, as for many other digital systems, complete testing is considered prohibitively expensive [IEEE, 1990].

The methods outlined above for the procurement of reliable hardware are all fault avoidance techniques. A complementary approach involves tolerating faults through the implementation of special design features. Fault tolerant techniques can be divided into those that detect faults and initiate recovery such as parity checking and watchdogs, and those that mask faults such as Triple Modular Redundancy (TMR) and error-correcting codes [Carter, 1985].

1.4.2. Reliable Software

Software faults (commonly called 'bugs') can arise from the specification, design, or coding process. Typically more than half the faults which are recorded during the software development originate in the specification [O'Connor, 1985]. This is mainly due to the use of natural language for documenting the 'non-technical' user requirements specification [Hitt & Webb, 1985]. Engineering principles are being proposed [Sommerville, 1985] to enable defect-free development and maintenance of software.

Software verification involves semantic and syntactic checks on the program code for programmer error, and structured walk-through checks for functional correctness. Black-box tests can be used to identify faults but the complexity of software often

prevents exhaustive checking due to prohibitive costs. The complexity of software testing can be reduced by adopting a modular code structure. Many methods have been proposed to assess acceptable test-set coverage for software [Musa et al, 1987] but they all are subject to the Dijkstra maxim 'testing reveals the presence of faults, not their absence' [Dijkstra, 1972].

Software can be manipulated to tolerate faults. Two well known approaches to fault tolerant software are N-Version Programming [Chen & Avizienis, 1978], and Recovery Blocks [Randall, 1975]. Both techniques rely on design diversity, the availability of multiple implementation of a specification, to tolerate faults. N-Version Programming requires the independent implementation of multiple, 'N', versions of the specification. These versions are processed in parallel with the same inputs. A voter collects the outputs and a majority decision made to select the perceived correct output. Theory implies high reliability for this method, but in practice the multiple program versions can share common mode failures [Eckhardt & Lee, 1988].

Recovery Blocks consist of a primary routine, which normally performs a task; an acceptance test which checks the primary routine result; and an alternative routine which is executed if the check fails. Unlike N-Version Programming where routine independence is assumed, Recovery Blocks require ensured independence between the primary routine, the acceptance test, and the alternative routine. Application of recovery blocks improves reliability. The degree of reliability can be enhanced by extending the number of independent alternative routines ensuring each acceptance test is also wholly independent.

Software data structures can also be manipulated so that they can tolerate the presence of faults. Taylor et al [1980] briefly outline this topic and propose fault tolerant structures for linear lists and binary trees.

1.5. Evaluating Controller Reliability

To evaluate the reliability of a microprocessor-based controller it is necessary to apply a 'systems approach'. The systems approach involves integrating the interdependencies of all sub-systems constituting a whole system. Microprocessor-based controllers consist of two entities; hardware and software. Many authors, including

Hitt & Webb [1985], and Ferrara et al [1989], integrate calculations for hardware and software reliability. However such reliability assessments do not involve any allowance for the internal interaction of hardware faults on software. Internal hardware/software interaction occurs across what is referred to as the 'interface'. To determine system reliability more accurately it is necessary to integrate assessments of hardware reliability, software reliability, and interface reliability.

1.5.1. Hardware Reliability

It is valuable to calculate the reliability of a hardware product for the duration of its 'useful' lifetime. Historical failure data which takes into account benign operating conditions and general age degradation is used to assess the expected lifetime of the hardware. Popular compilations of such data include the United States Air Force 'Reliability Prediction for Electronic Systems' (MIL-HDBK-217), and the United Kingdom British Telecom 'Handbook of Reliability Data' (HRD-4). Techniques for manipulating this data to reflect hardware architecture are well understood [Lala, 1985].

1.5.2. Software Reliability

Methods of establishing the reliability of software are still under development. Although many techniques have been proposed none have had the widespread acceptance given to the corresponding assessment of hardware reliability.

Assessments of software reliability usually involve the prediction of errors existing in the software. However, the reliability of the software depends not only on the existence of a fault but also its activation. Many authors have used Markov processes to model software reliability [Musa, 1987]. There are two types of Markovian software reliability model widely used; Poisson and binomial. The Poisson models assume an infinite number of faults in the software, whilst the binomial models assume a fixed number of faults. Both model types assume faults exist randomly within the software. Musa [1975] refined the basic Poisson model so that fault activation is a function of the time for which the software is executed. In reality, however, fault exposure is dependent on the fault location within the software and its associated probability of

activation by program execution. Littlewood [1981] attempts to model this situation with a binomial model that weights software faults according to the probability of their execution. Trachtenberg [1990] has recently reviewed and suggested a general theory of software reliability models based on Markovian processes.

The Markov software reliability models provide a valuable indication of the likelihood that a fault will be exposed during software execution. The hazard attributed to fault exposure can be further estimated by using *ad hoc* methods such as 'Fault Tree Analysis' (FTA) or 'Failure-Mode, Effects, and Criticality Analysis' (FMECA).

1.5.3. Interface Reliability

In software controlled digital systems, failures can occur which are difficult to diagnose as being due to the exposure of a hardware fault or software error. A distinction is not clear usually because the systems internal hardware/software interface has not been defined. The interface occurs within electronic devices such as processors and memories. For example, a fault in an individual cell on a memory device holding a program can cause what appears to be a software error. Memory devices are sometimes referred to as firmware to reflect their hardware/software interface. Other faults may occur on a data bus line with similar effect.

Permanent faults relating to interface reliability should be identified by the burn-in procurement of the hardware. However because of their limited duration, temporary faults are rarely located during the burn-in process. Assessment of the interface reliability requires knowledge of the occurrence of faults and errors they induce.

1.6. Low-Cost Enhancement of Controller Reliability

The reliability of microprocessor controllers can be enhanced by addressing the problem of faults introduced during procurement and operation. Techniques for procuring reliability have been briefly outlined. Operational faults are generated by component aging and transient disturbances in the working environment such as power supply fluctuations, electro-magnetic interference (EMI), and electro-static discharge (ESD) [Siewiorek & Swarz, 1982].

Transient disturbances are associated with temporary faults in digital systems. Unlike analogue or mechanical systems which tend to pass the effect of a transient disturbance as a temporary signal discrepancy whilst retaining overall function, digital systems are susceptible to malfunction in the presence of temporary faults because of their discrete state nature. Indeed it is becoming established that, even in 'benign' working environments, about 90% of microprocessor system failures can be attributed to temporary faults [Siewiorek & Swarz, 1982].

Control systems are often required to operate in 'harsh' industrial environments liable to produce transient disturbances. Although shielding can be employed to reduce the effects of transient disturbances on digital systems, their elimination is rarely possible [Horowitz & Hill, 1986]. The benefit of tolerating temporary faults induced by transient disturbances can be considerable. Industrial microprocessor controllers, however, are often developed within a limited budget which cannot support the redundancy incurred by many established fault tolerant techniques. This thesis approaches the topic of interface reliability, proposing a low-cost software-implemented fault tolerant technique for temporary hardware faults.

1.7. Thesis Preview

The topic of reliability for microprocessor-based controllers has been introduced with respect to the requirement for low-cost fault tolerance (the objective of the research presented in this thesis). Chapter 2 surveys literature investigating the fault-error-failure mechanism in microprocessor systems. The failure process is identified with malfunction, particular hazard being associated with the corruption of program flow. Current techniques to detect this class of fault are reviewed, but many require considerable expense to implement.

As a first step to developing new and more cost-effective techniques to detect program flow corruption, it is useful to consider the character of associated erroneous microprocessor behaviour. Chapter 3 presents a model for erroneous microprocessor execution. Performance parameters are evolved to show the benefit of implementing a detection capability together with a recovery mechanism. These parameters

include detection latency, reliability, Mean Time To Failure (MTTF), and availability. The model is applied to a selection of microprocessors commonly embedded within controllers, results are discussed in Chapter 4.

Microprocessor software displays various characteristics depending on the function of its implementation. Functional sections of code include program areas, data areas, and reserved memory mapped input/output areas. In addition, the microprocessor may have a proportion of its address space which is not populated by software. Whilst correct microprocessor operation executes instructions within the program area in a predictable manner, erroneous execution can invalidly interpret an instruction anywhere in the microprocessor address space in an unpredictable manner. Fault tolerant techniques whose implementation is based on software for detecting erroneous execution within each functional area of the microprocessor address space are presented in Chapter 5. In particular a novel technique for detecting erroneous program flow is proposed.

An algorithm for implementing the proposed fault tolerant technique in the program area is presented in Chapter 6. The technique involves manipulating the program code in order to strategically insert detection mechanisms. The mechanisms are designed to detect erroneous execution by identifying beforehand particular corrupted execution routes in the software. The algorithm is implemented in a software utility so that program code can automatically be given the detection capability. The software utility is called Post-programming Automated Recovery UTility (PARUT). PARUT is designed to be flexible, allowing the generation of fault tolerant code for a selection of microprocessors.

Several example programs are processed by the PARUT algorithm in Chapter 7 so that the performance of the fault tolerant technique can be assessed. Enhanced program code is evaluated by emulation and fault injection programmes. These experiments monitor the behaviour associated with corrupted patterns of erroneous execution. The information obtained from the experiments enables the detection performance of individual programs to be assessed.

Translators are used to generate significant amounts of software for microprocessor based controllers. The programmer has no control over the nature of the translator generated code. Chapter 8 identifies critical hazards which are not covered by the fault tolerant techniques outlined in Chapter 5. These hazards are associated with the catastrophic failures of cessation of processing and infinite execution loops. Techniques are proposed for the translator code generation process so that critical hazards are not produced. These techniques are influenced by the nature and structure of the target microprocessor instruction set.

Fault tolerant microprocessor design features are proposed in Chapter 9. These features facilitate rapid detection of corrupted program flow.

The final chapter reviews the thesis and draws conclusions on the research. Five appendices provide details of: microprocessor parameters applied in Chapter 4 to the model presented in Chapter 3; the design of a hardware unit associated with the software implemented fault tolerant technique proposed in Chapter 5; a code listing of the PARUT tool described in Chapter 6; the example programs evaluated in Chapter 7; and the papers published by the author relating to work presented in this thesis.

Chapter 2

TEMPORARY FAULTS: GENERATION, IMPLICATION, AND DETECTION

CHAPTER TWO

TEMPORARY FAULTS: GENERATION, IMPLICATION, AND DETECTION (A LITERATURE REVIEW)

2.1. Introduction

An industrial environment can be less than ideal for microprocessor-based controllers. In particular externally generated transient events can disrupt microprocessor operation. Erroneous microprocessor behaviour is associated with a degraded or lost control function, and the mal-operation of any equipment under the microprocessor based controller's supervision. Mal-operation of controlled equipment can be extremely hazardous because of the unpredictable nature of erroneous microprocessor behaviour.

This chapter discusses transient events that lead to temporary corruption of a microprocessor bus, register, or memory. Such corruption incurs no permanent hardware damage. The limited duration of temporary faults inhibits their detection. Without detection, and exercised by circuit action, temporary faults generate errors which can spawn other errors, the process terminating as either benign activity or catastrophic failure. The fault-error-failure mechanism is explored using the results of fault observations in real processor systems, fault simulations, and physical fault injection experiments.

Prolonged periods of erroneous behaviour increase the likelihood of generating a catastrophic failure. In order to prevent catastrophic failure, erroneous operation must be detected and appropriate recovery action initiated. Error detection is achieved by identifying characteristics of erroneous behaviour. Several techniques are reviewed which offer fault tolerance suitable for low-cost microprocessor based systems. The performance of the techniques is discussed in relation to the benefit of rapid error detection. Finally, assessing the reliability of a microprocessor systems adopting a fault tolerant technique is considered.

2.2. Faults and Their Implication on Microprocessor System Reliability

Assessing the reliability of a microprocessor based system involves evaluating the probability of failure which in turn is dependent on the fault-error-failure mechanism. As defined in Chapter 1, a fault is physical defect, an error is an activated fault, and a failure is classified as the deviation of system behaviour from that expected. The time interval between the occurrence of a fault and its manifestation as an error is called the *fault latency.* Similarly, the time interval between the occurrence of an error and the generation of a failure is called the *error latency.* Fault and error latencies are shown in Figure 2.1. The relationship between faults, errors, and failures is now explored.

Over the last decade, computer failure data has been collected for several continuously operational computer systems. Diagnosis of the data reveals temporary faults to be a significant cause of microprocessor failure. Collated computer failure data, see Table 2.1., shows temporary faults to account for between 93% and 98% of the induced computer system failures, the remaining failures being due to permanent faults. Furthermore, within the selection of computer systems, temporary faults have been observed to occur approximately every 40 to 330 hours during continuous operation. Temporary faults in digital devices are associated with electro-magnetic interference (EMI) [Liu & Whalen, 1988], electro-static discharge (ESD) [Bhar & Mahon, 1983], electrical noise [Shoji, 1987], ionizing radiation [Amerasekera & Campbell, 1987], and power supply fluctuations [Côrtes et al, 1986].

The manifestation of a temporary fault within a microprocessor based system is dependent on the susceptibility of its digital circuitry. Ball & Hardie [1969] report experiments which suggest that the probability of logic malfunction is dependent on the duration of the temporary fault. Typically, temporary faults only had a significant effect when they existed in excess of five clock cycles. Sequential logic was more susceptible than combinational logic, its probability of malfunction being in excess of 90% for temporary faults of 100 clock cycle duration.

Additionally, the miniaturisation of digital integrated circuits (scaling) makes them more vulnerable to many environmental effects [Russel & Sayers, 1989]. Scal-

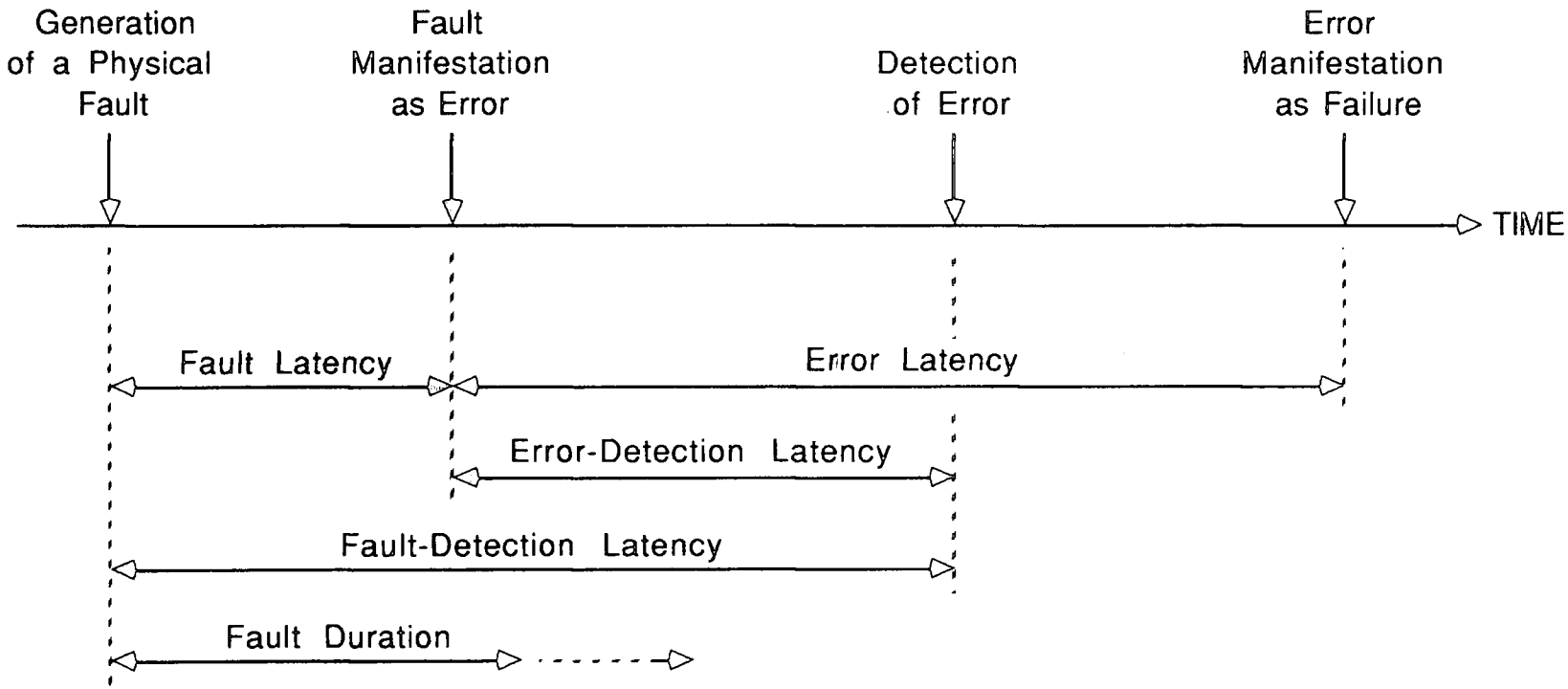


Figure 2.1. : Fault and Error Latencies

System	Source	Processor	Technology	Detection Mechanism	Hours Observed	Mean Time To Failure (Hours)		Temporary Faults (%)
						Temporary Fault	Permanent Fault	
CMUA	Fuller & Harbison, [1978]	PDP-10	ECL	Parity	5700	44	800-1600	94.8 - 97.3
Cm*	Siewiorek et al, [1978a]	LSI-11	NMOS	Diagnostics	15000	128	4200	97.0
C.vmp	Siewiorek et al, [1978a]	TMR LSI-11	NMOS	Crash	15000	97-328	4900	93.7 - 98.1
Telettra	Morganti et al, [1978]	UDET 7116	TTL	Mismatch	N/A	80-170†	1300	88.4 - 94.2
SLAC	Iyer & Rossetti, [1982]	N/A	N/A	Diagnostics	26000	58	2300	97.5‡
CMU-AFS	Lin, [1988]	MC 68010/20	NMOS	Diagnosis	212800*	201	6552	97.3
BG-Rig	Wingate & Preece, [1991]	TMR 8085	NMOS	Crash	9400	118	4100	96.3
BG-Rig	Wingate & Preece, [1991]	TMR 8085	NMOS	Crash	12300	154	3100	96.3

*Notes : † Reported by [McConnel, 1981]; ‡ From which 85% were recovered; * 13 applications monitored over 22 months.*

Table 2.1. : Failures Attributed to Temporary and Permanent Faults in Observed Microprocessor Systems

ing is associated with two operational characteristics on VLSI devices; lower operating power and higher processing speed. Lower operating power means that smaller power variations can generate a signal ambiguity or fault. Hence the severity of a transient disturbance required to induce a temporary fault is reduced. Higher processing frequencies enable smaller duration temporary faults to induce a logic fault.

A fault may have various effects depending on the state of the microprocessor system at the time of the fault and the duration of the fault. Errors are not generated when the fault duration is less than the fault latency. A primary error is produced when the duration of the fault is equal to the fault latency, the associated event probability being denoted by $Pr\{Error \mid Fault\}$. Subsequent errors, referred to as secondary errors, are generated in numbers that increase with the fault duration beyond the fault latency [Damm, 1988].

Errors can influence a microprocessor system in several ways. Errors can spawn further errors as modelled by Stiffler [1980] and Kopetz [1982]. Each of these errors can cause passive or active failure depending on whether or not they are dormant. Dormant errors, e.g. memory errors, have an error latency dependent on the access frequency of the corrupted memory locations. Hence, the microprocessor system application can have a major influence on determining whether or not an error leads to failure. Indeed Iyer & Rossetti [1986] and Woodbury & Shin [1990] both report evidence that a processor's workload can affect fault latency. Furthermore, the same error may have various effects on the microprocessor system depending on the timing of the error manifestation. The probability that an error generates a failure is denoted by $Pr\{Failure \mid Error\}$.

Reliability is defined in Chapter 1 as the probability of operating without failure, and can be expressed as,

$$R = 1 - Pr\{Failure\}. \tag{2.1.}$$

Inserting conditional probabilities, associated with the fault and error latencies, representing the fault-error-failure mechanism yields,

$$R = 1 - Pr\{Failure \mid Fault\}.Pr\{Fault\}. \quad (2.2.)$$

Techniques to improve microprocessor system reliability can be broadly divided into two groups: those that endeavour to prevent or reduce the generation of faults (fault prevention, to reduce $Pr\{Fault\}$), and those which attempt to intervene and prevent generated faults from causing system failure (fault tolerance, to reduce $Pr\{Failure \mid Error\}$ and $Pr\{Error \mid Fault\}$).

Fault prevention can involve the strategic implementation of two types of techniques [Anderson & Lee, 1982]. Firstly, *fault avoidance* can be employed to protect the controller from transient disturbances. This commonly involves 'shielding' the controller to obstruct the effects of a transient disturbance. Secondly, *fault removal* can be applied to identify weak design or components within the microprocessor-based controller. This process is commonly referred to as 'screening'. Rectifying the weaknesses should reduce the susceptibility of the controller to the effects of transient disturbances.

Anderson & Lee [1982] identified four operations to be complete in order for a system to be fault tolerant.

- error detection,
- damage assessment,
- error recovery, and
- fault treatment and continued system service.

Once an error is detected it is necessary to identify and isolate the damage incurred. Then the system must be restored to a valid operational state in order to prevent recurring errors evoking system failure. Finally, any damage must be repaired and the original system operation re-initiated. Temporary faults do not incur any physical damage and hence on completion of the fault tolerant process, assuming the temporary fault has terminated its existence, the system is returned to complete working order.

The duration of the error until its detection is called *error-detection latency.* The sum of the fault latency and error-detection latency is referred to as the *fault detection latency* [Damm, 1988], see Figure 2.1.

Equation (2.2.) denotes two levels at which fault tolerant techniques can be applied: the circuit (or gate) level $Pr\{Error \mid Fault\}$, and the functional (or component) level, $Pr\{Failure \mid Error\}$. Temporary faults are extremely difficult to detect because of their short duration. The overhead in providing a detection capability for individual faults in a VLSI device at circuit level is considered by Mahmood & McCluskey [1985] to be prohibitive. Some of the faults generated will in any case be benign and hence not require detection. Nevertheless, other faults, exercised by circuit operation, can generate errors. In order for a system to tolerate such faults it is necessary to detect their associated errors before they in turn induce system failure. Fortunately, there appears to be a good correlation between the circuit level and behaviour level reaction of common VLSI design elements, such as Arithmetic Logic Units (ALU) and multiplexers, to the effects of faults [Chakraborty & Ghosh, 1988] which suggests that functional level fault tolerance will be acceptable for most fault tolerant systems.

2.3. Erroneous Operation of Microprocessor Systems

The impact of faults upon microprocessor operation has been the subject of much research. Iyer uses fault simulation experiments, see Table 2.2., to investigate the susceptibility of generating and the likelihood of propagating functional element errors. Memory, arithmetic logic units (ALU), and multiplexers are found to be, in descending order, the most susceptible functional elements to error generation and propagation. The remaining fault simulation experiments in Table 2.2., together with the physical fault injection experiments listed in Table 2.3., investigate the effect of such errors on software operation.

Software execution errors generated by faults are diverse, but can be divided into two general groups: data flow errors and program flow errors. Both are affected by fault latencies with a bimodal distribution, i.e. there are two or more distinct classes of error manifestation [Arlat et al, 1990] [Czeck & Siewiorek, 1990]. Faults, such

Source	Target Processor	Method	Goal
McGough & Swern, [1981, 1983]	AMD 2901	Emulation	Coverage Measurement
Marchal & Courtois, [1982]	MC6800 & MC68000	Simulation	Detection Time
Li et al, [1984]	SBR9000	Emulation	Coverage Measurement
Yang et al, [1985]	iAPX 432	Emulation	Coverage of TMR
Lomelino & Iyer, [1986]	AMD 2901	Simulation	Error Propagation
Duba & Iyer, [1988]	HS 1602	Simulation	Error Propagation
Kim & Iyer, [1988]	AMD 2901	Simulation	Error Propagation
Segall et al, [1988]	Intel 80186	Simulation	Concept Study
Choi et al, [1989]	HS 1602	Simulation	Error Propagation
Czeck & Siewiorek, [1990]	Intel 80186	Simulation	Detection Time & Coverage

Table 2.2. : Fault Emulation/ Simulation Experiments.

Source	Target Processor	Method	Goal
Crouzet & Decouty, [1982]	Intel 8080	Hardware Pin	Methodology Study
Liu, [1982]	Z8	Ion Radiation	System Evaluation
Schmid et al, [1982]	Z80	Hardware Pin	Error Detection Evaluation
Lala [1983]	FTMP	Hardware Pin	System Evaluation
Cusick et al, [1985]	Z80 & NSC-800	Ion Radiation	System Evaluation
Damm, [1986]	MC68000	Power Supply	Error Detection Evaluation
Schuette & Shen, [1986]	MC68000	Hardware Pin	Error Detection Evaluation
Chillarege & Bowan, [1989]	IBM 3081	Memory Corruption	Large System Failures
Gunneflo et al, [1989]	MC6809	Ion Radiation	Error Detection Evaluation
Arlat et al, [1990]	MC68000	Hardware Pin	Methodology Study
Madeira et al, [1990]	Z80	Hardware Pin	Error Detection Evaluation

Table 2.3. : Fault Injection Experiments.

as stack corruption, are dormant requiring particular processing for their activation whilst other faults generate 'fast' failures. Nineteen percent of faults injected into a Motorola 6809 microprocessor system through ion bombardment [Gunneflo et al, 1989], and 22% of faults injected into an IBM 3081 processor system through memory corruption [Chillarege & Bowan, 1989], produce dormant faults.

Gunneflo's experiment also reports 78%, 17%, and 5% of the injected faults to generate program flow errors, data flow errors, and other consequences respectively [Gunneflo et al, 1989]. Experiments that physically inject faults on processor package pins, Schmid et al [1982] and Schuette & Shen [1986], support these results with 63% and 78% of faults generating program flow errors in Zilog Z80 and Motorola 68000 microprocessor systems respectively. Further, McGough & Swern [1981] report over half the logical faults inserted in a simulation of a AMD 2901 processor system to generate 'wild' branches, i.e. program flow errors. Caution should be exercised when comparing the significance of data and program flow errors in the fault insertion experiments because the experiments use different microprocessors, different fault insertion methods, and different fault locations.

2.3.1. Data Flow Errors

Data flow errors are generated by corruption or incorrect processing of data elements resident in data structures or instruction operands. Erroneous data can produce 'unreasonable' as well as failure conditions [Damm, 1988]. Data flow errors do not disturb the program flow and hence acceptance tests can be embedded within the program to check for bad data.

2.3.2. Program Flow Errors

Program flow errors can be initiated in microprocessor systems by the incorrect identification of a memory location as containing an instruction [Marchal & Courtois, 1982], or corruption of an instruction itself [Carpenter, 1989]. There are three main mechanisms by which faults can generate erroneous program flow. Firstly, corrupted opcodes may have a different instruction length. Carpenter [1989] demonstrates this effect for the Motorola 68000 microprocessor. A change in the instruction length

forces incorrect interpretation of the memory location for the next instruction opcode. Secondly, corruption of a branch or jump instruction operand will alter the location of the next instruction opcode to be executed. Finally, registers in the microprocessor specifying the address of the next instruction opcode may be directly corrupted. A SBR9000 processor fault simulation, [Li et al, 1984], reports 73% of program flow errors to be generated in this way. Either of these three mechanisms effectively initiates execution of an unknown program. The nature of ensuing execution is dependent on the code content of memory under interrogation. The character of this execution is haphazard and not usually in sympathy with the organized execution associated with the application program. This behaviour leads to the malfunction of the controller. Particular hazard is attributed to such occurrences because of the implications of their unpredictable effect on controller operation.

Once incorrect memory locations are accessed for instruction opcodes then there is a high probability that following instruction opcodes will also be incorrectly identified [Marchal & Courtois, 1982]. As erroneous program flow progresses, processing may further corrupt memory containing the original software. Erroneous program flow can terminate naturally via re-synchronization. Re-synchronization involves re-establishing identification of instruction opcodes within the original software application program. A physical fault injection experiment reports 75%, 6%, and 19% of program flow errors diverge permanently from the correct program (program crash), diverge temporarily from the program flow (re-synchronization), and are dormant (stack errors, etc) respectively. Tests cannot be embedded within the application program to identify opcode corruption because the application program is no longer executed.

Sosnowski [1986a] considers steady state outcomes of erroneous behaviour associated with microprocessor failure. He develops models for false loops, traps, and deadlocks. False loops and deadlocks are essentially infinite execution loop phenomena which have also been modelled by Halse & Preece [1987]. Deadlocks involve the termination of execution when the processor enters a 'wait' state. Halse & Preece

[1985] and Sosnowski [1986b] investigate characteristics of erroneous behaviour relating to the influence of different microprocessor instruction sets and address space utilization.

2.4. Assessing Error Detection Techniques for Microprocessor Systems

Hardware and software fault tolerant techniques are briefly outlined in Chapter 1. Many techniques, however, may be unsuitable for microprocessor controllers because the costs of their application exceed the controllers budget. This section reviews low-cost fault tolerant techniques for microprocessor controllers requiring high reliability.

2.4.1. Watchdog Timers

One of the most basic techniques for checking the operation of a microprocessor-based system is the use of a watchdog timer [Connet et al, 1972], [Ornstein et al, 1975]. The system is designed such that, under normal operation, program execution signals the watchdog timer within a specified time interval. The signal presets the timer to its initial value. The timer generates an error if no preset signal is forthcoming during the specified time interval. On receiving the error signal from the watchdog, the system initiates suitable recovery action. Typically this involves re-establishing a correct set of operating parameters. Watchdogs incur a small switching overhead and no performance degradation is incurred directly upon the executing software.

Watchdog timers, however, may have a hazardous error-detection latency. Consider a processor operating at 8 MHz, whose mean instruction processing time is 100 clock cycles, implementing a watchdog with a 100 ms interval. If a malfunction occurs in the middle of this interval then approximately 4000 instructions can be processed erroneously before the malfunction is detected. During the malfunction, processor operation is uncontrolled and may have hazardous implications for the processor activity. Therefore, other techniques providing fault detection are being developed.

2.4.2. Capability Checking

Lu [1980] was one of the first to propose what is now commonly referred to as the 'smart' watchdog. These units are based on an additional processor to provide

a monitoring capability, facilitating faster detection without the high cost associated with fault masking techniques. Mahmood et al [1983] proposed a smart watchdog to check algorithm-level assertions about executing software. Namjoo & McCluskey [1982] suggested a scheme called 'capability checking' implemented by a smart watchdog that identifies illegal operations and memory accesses. Such a unit would detect system malfunctions as well as prevent memory mutilation by erroneous behaviour. Marchal & Courtois [1982] applied a selection of capability checks, whilst Schmid et al [1982], referring to capability checking as 'abstraction verification', extend the test set and estimate fault detection through direct fault simulation. Smart watchdogs can be applied to modern microprocessors that implement co-processors and caches [Saxena & McCluskey, 1990]. Mahmood & McCluskey [1988] survey the use of smart watchdogs.

Capability checks, see Table 2.4., implemented together create reliable computer systems as demonstrated by Schmid et al [1982], Gunneflo et al [1989], and Madeira et al [1990] with 88%, 79%, and 75% fault detection respectively. The variation in fault detection is due to different selection of the capability checks implemented by each system, and the method of fault insertion when evaluating the microprocessor systems fault tolerance. Collectively applying capability checks provides both program and data flow error detection. However, implementing all these techniques can be complex so another simpler alternative is being explored by researchers. It involves detecting data flow errors by placing reasonableness checks in the software [Damm, 1988] (this includes Recovery Blocks, and N-Version Programming), and implementing a monitoring scheme to directly verify the program flow of the application processor.

2.4.3. Program Flow Monitoring

Program flow monitoring schemes are typically based, to some extent, on control flow graphs. The graphs consist of linked nodes. Each node represents a sequence of instructions performing some task, and each link represents the transition conditions, i.e. status information. Lu [1982] proposed a scheme called 'structural integrity

Capability Checks
a) incorrect sequence of instructions
b) branch to invalid destination
c) fetch illegal instruction
d) fetch an opcode from a none opcode address
e) invalid read within permitted memory
f) invalid write within permitted memory
g) access to memory outside permitted memory area
h) watchdog timer

Table 2.4. : Capability Checks

checking' involving the generation of a tag for each task. These tags are checked during execution to verify correct operation. Lu does not check transition conditions. Yau & Chen [1980] ensure each task does not have any inherent loops, hence, preventing the possibility of infinite erroneous execution loops without potential detection. They also implement verification of transition conditions between tasks.

Task tags assigned values based on cyclic encoding of their instruction sequences are called 'signatures'. Two techniques implementing signatures, Path Signature Analysis [Namjoo, 1982] and Signatured Instruction Streams [Shen & Schuette, 1983] embed precomputed signatures into the application program. During program execution, special circuitry re-computes the signature and compares it with the embedded code signature, any ambiguity signalling detection of erroneous program flow. Both techniques impose a performance and code overhead. Schuette & Shen [1986] have implemented an embedded signature technique. The dedicated circuitry required 3947 gates and 5435 bytes of memory, a hardware overhead of approximately 38% compared to the gate count of the host Motorola 68000 application processor. The memory overhead is incurred by embedded tags in the application program: typical overhead estimates range between 10 and 20% [Wilken & Shen 1987]. Finally, pseudo-branches, required by the implementation so that correct execution by-passes embedded signatures, are estimated to reduce application program performance by 10%. The technique implemented in a Motorola 68000 processor system is reported to have a mean detection latency of less than 100 μs, the maximum latency being 3.8 ms. This is a considerable improvement on the detection latency expected from watchdogs. Schuette & Shen [1986] and Segall et al [1988] report a 98% and 94% coverage of program flow errors respectively. Wilken & Shen [1987] review the mechanism of several signature monitoring techniques that embed signatures in the application software.

Namjoo [1983] proposes a smart watchdog to compute run-time signatures independently to verify application processor behaviour. This technique does not incur the performance or code overhead associated to earlier schemes. Eifert & Schuette [1984] refine the technique, replacing the smart watchdog with dedicated circuitry.

These techniques whilst not degrading system performance or requiring extra memory, do require an additional hardware unit. Smart watchdogs introduce approximately 100% redundancy by duplicating the number of processors. Replacing the watchdog with dedicated circuitry implies that the design is not directly applicable for use with different microprocessor types.

2.4.4. Hazards Associated with Error Detection Techniques

It should be noted that those fault tolerant techniques implementing hardware redundancy will also be susceptible to the effects of faults induced by the environment. In particular, hazard is associated with those detection techniques such as the smart watchdogs that use a microprocessor to monitor a microprocessor. Duba & Iyer [1988] and Choi et al [1989] in their fault simulations identify the watchdog element of their microprocessor system to be significantly vulnerable to temporary fault generation, and diagnose a critical fault propagation path between the control unit and the watchdog. The reliability of a microprocessor-based controller implementing a watchdog device can be seriously undermined if the detection capability of the watchdog is lost. Damm [1988] refers to this occurrence as the 'doomsday' syndrome.

2.4.5. A Novel Error Detection Technique

This thesis proposes an alternative technique based on the self-detection of program flow errors by erroneous execution. Potential program flow errors within the software are identified and the code structure enhanced by the strategic placement of detection mechanisms. These mechanisms can only be activated by erroneous program flow. The technique does not inherently require additional hardware. The only overhead is the extended code requirement, and the associated additional execution to by-pass the inserted detection mechanisms. The code extension, and the degradation to application program performance by by-pass operations is comparable with that required by the embedded tags and pseudo-branches reported for an implementation of the embedded signature technique [Schuette & Shen, 1986], [Wilken & Shen, 1987].

2.5. Reliability Evaluation

The reliability of microprocessor systems implementing fault masking can be assessed using architectural analysis. However, this method cannot be used in microprocessor systems implementing other fault tolerant techniques, such as those discussed in this chapter because of the uncertainty of fault detection.

An alternative reliability evaluation which can be adapted to the fault tolerant techniques discussed in this chapter is given by Schuette & Shen [1986]. They propose the following estimation for reliability of a microprocessor-based system, $R_s(t)$, employing signature monitoring,

$$R_s(t) = \left[R(t).R^F(t)\right] + \left[(1 - R(t)).R^F(t).E\right]. \tag{2.3.}$$

where the reliability of the microprocessor-based system $R(t)$ is the expectation of an error occurring in the microprocessor before time t, $R^F(t)$ is the expectation of an error occurring in the additional circuitry required by the fault tolerant mechanism before time t, and E is the error coverage of the employed detection mechanism.

Re-arranging equation (2.3.) gives,

$$R_s(t) = R^F(t)[R(t) - (1 - R(t)).E]. \tag{2.4.}$$

The correct operation of the system is dependent on avoiding the 'doomsday' syndrome discussed earlier, for the detection mechanism. Therefore the reliability of the system, $R_s(t)$, is directly dependent on the reliability of the detection mechanism, $R^F(t)$. An additional constraint on system reliability is the sum of the probability that the processor is working correctly, $R(t)$, with the probability of fault coverage by the detection mechanism when the processor is not working correctly, $(1 - R(t)).E$.

Reliability engineers commonly use the complementary reliability parameter Mean Time To Failure (MTTF) for failure rates; increases in the event rate reduce the MTTF expectations. Event rates are, as discussed earlier in this chapter, dependent on the occurrence of transient disturbances and the susceptibility of a system to this disturbance. It cannot be assumed that environmental conditions are stable – the

mean rate of occurrence of transient disturbances may approximate to a constant, but it should be recognized that occurrences may cluster. Clustered occurrences are known as 'bursts', and within the C.vmp microprocessor system approximately 25% of observed fault events were bursts [McConnel & Siewiorek, 1978].

Variations in the event rate will alter the expected reliability for individual applications, and hence limit the usefulness of equations like (2.3.) when evaluating system reliability. However, equation (2.3.) can be applied to many other fault tolerant techniques, implemented in a simple single processor system, to provide a comparative index of their effectiveness.

2.6. Summary and Conclusions

Temporary faults have been diagnosed as causing between 15 and 50 times more failures in microprocessor systems than permanent faults. Reliability engineers attribute the generation of many temporary faults to the occurrence of transienten-vironmental disturbances.

Microprocessor-based controllers are often required to operate in harsh environments where transient disturbances are a regular hazard. Prevention techniques involving screening and shielding can be applied in an attempt to eradicate the effects of transient disturbances on microprocessor controllers. In practice these techniques, however, only reduce the problem.

Temporary faults are difficult to detect because of their limited duration. Furthermore, the errors they generate can induce microprocessor malfunction and this may lead to erroneous operation of equipment under directives from the controller. Equipment operation may be haphazard and pose a danger in particular applications.

Microprocessor malfunction should be detected rapidly to reduce the hazard of erroneous equipment operation. It is with this purpose that fault tolerant techniques have been developed. Fault tolerant techniques enable digital systems to isolate and repair the effects of temporary faults before restoring the application function. Program flow errors are identified as having particular influence on the character of

microprocessor malfunction. A selection of techniques applicable to microprocessor-based controllers are reviewed. In particular, techniques are discussed which incur a low system overhead.

Assessment of the fault tolerant techniques reviewed in this chapter involves inserting faults into a microprocessor system and monitoring its response. These assessments only reflect the efficiency of the implemented fault tolerant techniques. A major number of observed computer system failures are attributed to temporary faults. Assessments of microprocessor system reliability should, therefore, take this class of fault into account. Reliability assessments should involve information including knowledge of the susceptibility of the processor to transient disturbances, and the likelihood of such disturbances in the application environment.

In summary, temporary faults can be responsible for a significant number of microprocessor system failures. Fault prevention techniques cannot guarantee the eradication of all faults. It is therefore pertinent to incorporate fault tolerant features into the system design. Effective fault tolerance can be implemented at low-cost. For microprocessor systems with safety applications, developed within limited financial budgets, these techniques can provide highly beneficial and cost-effective reliability.

Chapter 3

MODELLING ERRONEOUS MICROPROCESSOR BEHAVIOUR

CHAPTER THREE

MODELLING ERRONEOUS MICROPROCESSOR BEHAVIOUR

3.1. Introduction

This chapter investigates microprocessor behaviour with particular regard to fault conditions. Temporary hardware faults may disrupt software processing and induce erroneous execution. The event initiating erroneous behaviour is defined. A model is proposed to simulate erroneous microprocessor behaviour. This model is developed for the von Neumann class of microprocessor which has dominated processor design over the last thirty years. Erroneous behaviour is investigated. A facility for detecting erroneous execution is introduced into the model. The efficiency of detection is examined with respect to the latency between initiation and detection of erroneous behaviour. A stochastic reliability model is proposed to assess the effect of software disruption on microprocessor performance. A method is developed for calculating Mean Time To Failure (MTTF) of the microprocessor system. Availability is also determined under the assumption that the processor has a resident recovery routine in its memory. MTTF and availability are common engineering measures of reliability. Hence the modelled reliability for a microprocessor can be compared to other device reliabilities.

The models presented are robust, not relying on the features of any microprocessor(s). Model application allows the analysis and comparison of a wide selection of microprocessors.

3.2. Initiating Erroneous Microprocessor Behaviour

Erroneous microprocessor behaviour is considered to occur when a temporary fault manifests itself as a control flow failure. Loss of the correctly operating control flow will cause the microprocessor to mis-interpret its software with the hazard of malfunction. The ensuing behaviour comprises of propagating erroneous execution with a progressively increasing likelihood of catastrophic failure.

Any control flow failure will be reflected by an erroneous target entry in the microprocessor's program counter, and hence, the following assumption is made.

Assumption (1) : The event initiating erroneous microprocessor behaviour is that of program counter corruption.

The program counter is considered as a single register used to locate instructions through the whole address space. The nature of the program counter corruption is not known. The event initiating erroneous behaviour may have had a variety of sources including stack pointer corruption, bus-line transients, and memory bit-flips. It is assumed that all bits in the program counter are equally susceptible to error.

Assumption (2) : The contents of the microprocessor program counter are corrupted randomly by the event initiating erroneous behaviour.

These assumptions enables a mathematical model, based on probability theory, to be developed for erroneous microprocessor behaviour.

3.3. Erroneous Behaviour

Consider erroneous behaviour to be initiated by random corruption of the program counter. This event produces a jump in the control flow of the existing software to a random location in the address space of the microprocessor. This random jump is termed the *Initial Erroneous Jump* (IEJ). Erroneous execution then commences. The data contents of the memory at the new location will be executed as if they were instruction codes. Erroneous execution will take place in a linear fashion until the execution of a further jump instruction causes a *Subsequent Erroneous Jump* (SEJ). Repeated periods of linear erroneous execution interspersed by SEJs follow until terminated either by catastrophic failure or system recovery. This process is shown in Figure 3.1., where the execution flow through the address space is shown as a stream of linked periods of linear erroneous execution.

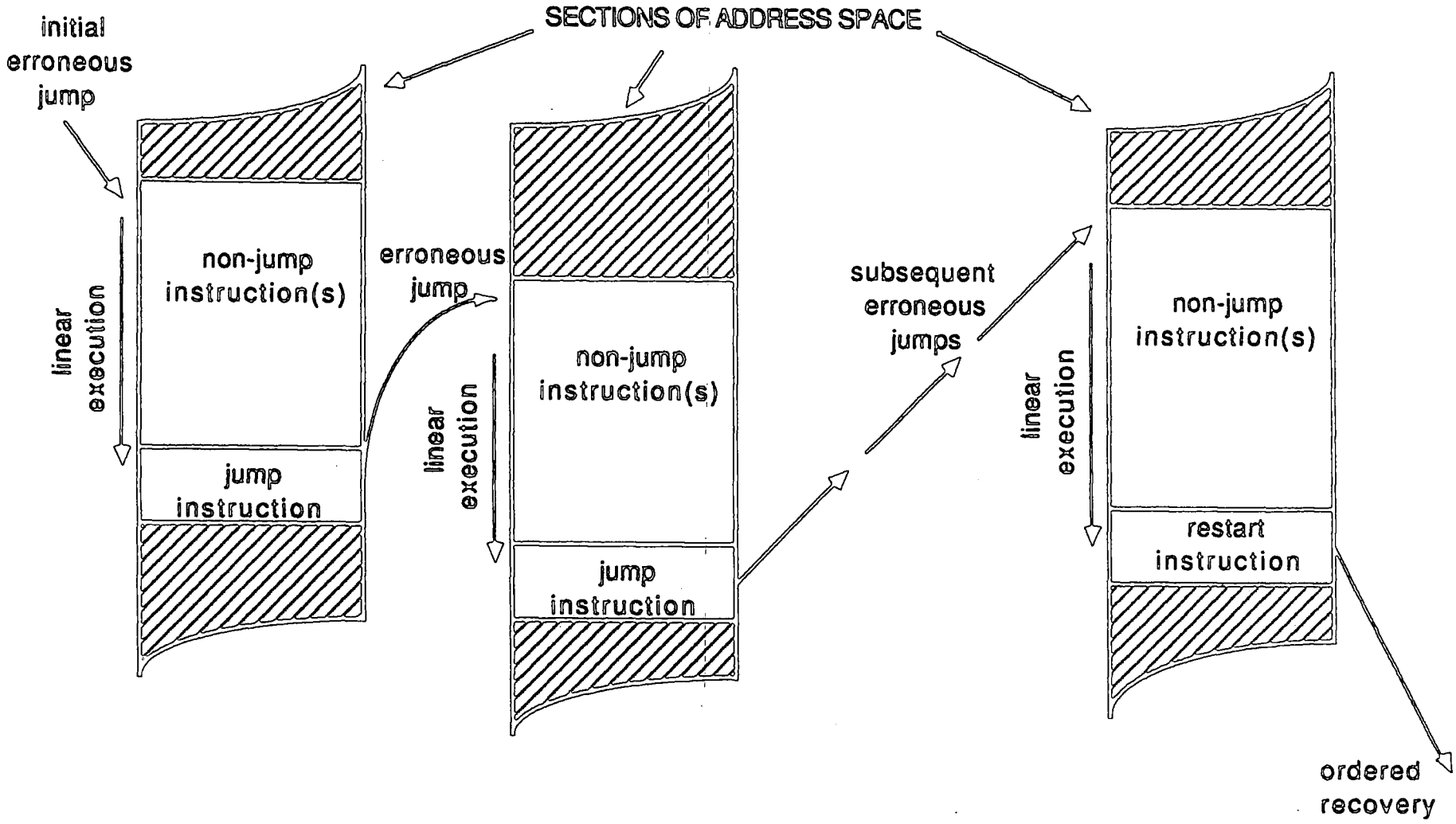


Figure 3.1. : Microprocessor Erroneous Behaviour.

3.4. Erroneous Execution

Erroneous execution consists of a sequence of execution states, each state representing the operation of an instruction. Execution states can be categorized with respect to the nature of their outcome. Halse [1984] identified the state outcomes listed below.

Non-Jump :	leads to the program counter pointing to the next instruction in the address space.
Restart :	leads to a jump to a predefined location in the address space.
Unspecified Jump :	leads to a jump to a new location in the address space determined by volatile memory contents.
Return :	leads to a jump to an address held in a stack.
Stop/Wait :	leads to a cessation of processing ; and requires an interrupt or hardware reset to exit from this state.

Restart outcomes are usually generated by interrupts or exceptions. The restart outcome vectors execution to a location predefined by the microprocessor architecture. A recovery routine can be placed at the restart outcome vector target. Hence for controlled recovery, a restart outcome defines erroneous behaviour detection and an ordered return to a recovery routine.

A model for erroneous execution is shown in Figure 3.2. The model shows erroneous microprocessor behaviour being entered by program counter corruption (IEJ). The cascading sequence of state outcomes throughout erroneous execution can be traced. Successive 'non-jump' state outcomes will produce a period of linear erroneous execution. A restart outcome can be used to provide detection of erroneous execution. Any of the remaining state outcomes, including a stop/wait outcome, are defined as generating further erroneous jump (SEJ) in the erroneous execution.

In a particular make of microprocessor, not all possible instruction bit patterns are necessarily defined as instructions. Illegal, or 'undefined' instructions can, when

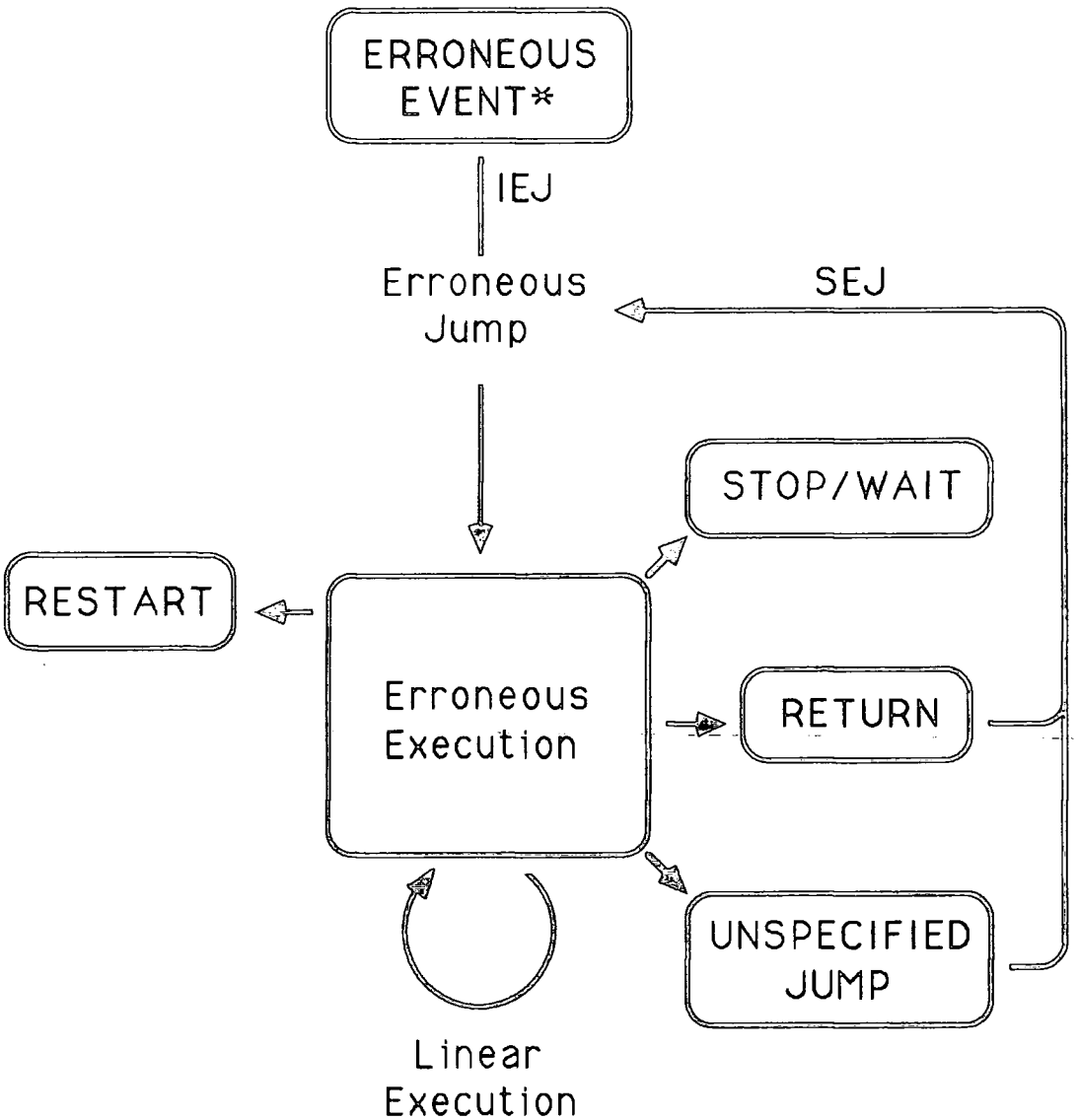


Figure 3.2. : Erroneous Execution Model
(* manifested transient fault)

executed, result in any one of the state outcomes defined above. The actual state outcome depends on the particular microprocessor die, as manufacturers are not obliged to ensure that every batch produces the same operation. In some machines, such as the Motorola 68000 and Intel 80386, the execution of all undefined instructions is specified as an exception (software interrupt) and hence will produce a restart state outcome.

3.5. Halse Execution Model

This section briefly reviews the foundations of a model for microprocessor operation proposed by Halse [1984]. The model analyses erroneous microprocessor operation.

A statistical model of erroneous execution is made using the assumption that the memory contents, throughout the address space, have a distribution that does not change. This clearly does not reflect the memory content distribution for real microprocessor based systems. Their distribution will vary through the memory map dependent on the utilization of locations. Nevertheless, the model does enable the identification of some general characteristics of erroneous execution.

As a result of erroneous behaviour, erroneous execution will interpret some location in the address space as an instruction. This results in one of two outcomes. Either an erroneous jump is generated which transfers control to another part of the address space; or no jump occurs and control passes onto the next logical location.

Let the probability of an instruction execution yielding a 'jump' or 'non-jump' outcome be P_J and P_{NJ} respectively. Hence by definition,

$$P_J = 1 - P_{NJ} \tag{3.1.}$$

It follows that the probability of terminating a period of linear execution on the k^{th} instruction (i.e. generate an erroneous jump) is given by

$$P_J(k) = P_{NJ}^{(k-1)}.P_J, \quad \text{where } k \geq 1. \tag{3.2.}$$

When evaluating a microprocessor's erroneous behaviour, it is more realistic to use effective instruction outcome distributions rather than instruction outcome distributions based upon instruction set definitions. It is recognized that some instructions have different outcomes dependent on some conditional test. In particular it is noted that conditional branch instructions can be paired, such that groups of two instructions covered a condition and its complement. Hence each pair of conditional branch instructions, such as a 'branch if zero' and 'branch if not zero', can be treated as if it were a single jump instruction and a single non-jump instruction. Halse [1984] assumes conditional instructions to have a 50% chance of occurring. Although this is strictly not true for individual conditions, the overall treatment of conditional instructions in this manner is considered valid.

Let Ne_J be the effective number of address space locations that when interpreted as an instruction generate a 'jump' outcome instructions. Let N_L be the number of address space locations that could be interpreted as an instruction.

Then:

$$P_J(1) = \ P_J = \frac{Ne_J}{N_L}, \tag{3.3.}$$

and,

$$Ne_J = Ne_{RN} + Ne_{RT} + Ne_{S/W} + Ne_{UJ} \tag{3.4.}$$

where Ne_{RN}, Ne_{RT}, $Ne_{S/W}$, and Ne_{UJ} are the effective numbers of 'return', 'restart', 'stop/wait', and 'unspecified jump' outcome instructions in the available address space.

The probability that termination of a period of linear execution results in a particular outcome $P_x(k)$, is dependent on the proportion of that type of instruction in the set of 'jump' instructions. Collecting equations (3.2.), (3.3.), and (3.4.) gives,

$$P_x(k) = \begin{cases} 0, & \text{when } k = 0, \\ \frac{Ne_x}{Ne_J}.P_J(k), & \text{when } k \geq 1. \end{cases} \tag{3.5.}$$

where the subscript x denotes the respective jump outcomes; $x \in \{RT, UJ, RN, S/W\}$ represents restart, unspecified jump, return, and stop/wait.

3.6. Hybrid Execution Model

A hybrid model is proposed here facilitates further investigation of erroneous microprocessor behaviour. Erroneous behaviour has two phases of execution; linear execution following an IEJ, and linear execution following an SEJ. The hybrid model enables the examination of the characteristics associated with each of the two patterns. In particular, three mechanisms have been identified that terminate linear erroneous execution.

a) Another period of linear erroneous execution is initiated: an 'unspecified jump' or 'return' state is entered.
b) Processing stalls: a 'stop/wait' state is entered.
c) Detection of erroneous execution: a 'restart' state is entered.

This section initially models the periods of linear erroneous execution associated with each of the two patterns of behaviour. These are then developed to investigate the probability of further periods of linear erroneous execution being initiated, stalled, or detected.

3.6.1. Linear Erroneous Execution

Both restart and stop/wait erroneous jump outcomes terminate erroneous execution. Unspecified jump and return outcomes initiate a new period of linear erroneous execution. Let $P_{J'}(k)$ represent the probability of the kth instruction processed yielding a jump outcome, other than a restart or stop/wait, following an erroneous jump. The subscript J' represents the jump outcomes; unspecified jump, and return.

$$P_{J'}(k) = \sum_{x \in \{UJ,RN\}} P_x(k), \quad \text{where } k \geq 0. \tag{3.6.}$$

Logical processing errors such as 'divide by zero' can cause premature completion of an instruction's execution, generating an otherwise unexpected restart outcome within a fraction of the clock cycles normally required by the instruction. These errors have different influences on each of the two phases of erroneous behaviour. An IEJ may not initiate erroneous execution but fire what is considered by the model

to be an immediate restart outcome. The same assumption is made for an SEJ such that the SEJ instruction is considered to fire a restart rather than a jump outcome. Let β and γ be the respective proportion of IEJs and SEJs firing a restart outcome in this way, such that $0 \leq \beta \leq 1$ and $0 \leq \gamma \leq 1$.

Consider the execution model probability for the outcome of the *kth* processed instruction, equation (3.5.). Let $P_x^{IEJ}(k)$ and $P_x^{SEJ}(k)$ be the probability density functions for the execution outcome of the *kth* processed instruction. The subscript x denotes the class of outcome; $x \in \{RT, UJ, RN, S/W\}$ representing restart, unspecified jump, return, and stop/wait outcomes respectively. The superscript IEJ or SEJ denotes execution following an IEJ or SEJ respectively.

Evaluating erroneous execution following an IEJ when $k = 0$;

$$P_x^{IEJ}(k) = \begin{cases} 0, & \text{where } x \in \{UJ, RN, S/W\}, \\ \beta, & \text{where } x \in \{RT\}. \end{cases} \tag{3.7a.}$$

and when $k \geq 1$;

$$P_x^{IEJ}(k) = \begin{cases} (1-\beta).[(1-\gamma).P_x(k)], & \text{where } x \in \{UJ, RN, S/W\}, \\ (1-\beta).\left[P_x(k) + \gamma.P_{J'}(k)\right], & \text{where } x \in \{RT\}. \end{cases} \tag{3.7b.}$$

Evaluating erroneous execution following an SEJ when $k = 0$,

$$P_x^{SEJ}(k) = 0, \qquad \text{where } x \in \{RT, UJ, RN, S/W\}, \tag{3.8a.}$$

and when $k \geq 1$,

$$P_x^{SEJ}(k) = \begin{cases} (1-\gamma).P_x(k), & \text{where } x \in \{UJ, RN, S/W\}, \\ P_x(k) + \gamma.P_{J'}(k), & \text{where } x \in \{RT\}. \end{cases} \tag{3.8b.}$$

Let $P_J^y(k)$ be the probability of terminating a period of linear erroneous execution where the subscript x denotes the class of outcome; $x \in \{RT, UJ, RN, S/W\}$ representing restart, unspecified jump, return, and stop/wait respectively. The superscript

y denotes execution following an erroneous jump; $y \in \{IEJ, SEJ\}$ representing IEJ and SEJ respectively.

$$P_J^y(k) = \sum_x P_x^y(k), \tag{3.9.}$$

The probability that k instructions have been linearly processed in the current phase of linear execution is evolved from equation (3.9.),

$$P_L^y(k) = 1 - \sum^k \{P_J^y(k)\}, \quad \text{where } k \geq 0, \text{ and } y \in \{IEJ, SEJ\}. \tag{3.10.}$$

The mean number of instructions expected to be executed during each phase of erroneous behaviour, I, is given by,

$$I = E[K] = \sum_k k.P_J^y(k), \quad \text{where } k \geq 0, \text{ and } y \in \{IEJ, SEJ\}. \tag{3.11.}$$

where K is the random variable of the probability density function, defined by equation (3.9.), $P_J^y(k)$.

If the probability of a jump in either of the patterns of erroneous behaviour is zero then the number of instructions processed during linear erroneous execution is infinite. This, of course, assumes that repeated passes of execution through the address space due to program counter overflow, are considered as a single period of linear erroneous execution.

3.6.2. Propagating Further Periods of Linear Erroneous Execution

Periods of linear erroneous execution are propagated when erroneous behaviour generates a SEJ. The probability of a SEJ outcome on the kth processed instruction during either of the two patterns of linear erroneous execution is developed from equation (3.6.),

$$P_{J'}^y(k) = \sum_x P_x^y(k), \quad \text{where } k \geq 0 \tag{3.12.}$$

where the subscript x denotes the class of outcome; $x \in \{UJ, RN\}$ representing unspecified jump, and return respectively. The superscript y denotes execution following an erroneous jump; $y \in \{IEJ, SEJ\}$ representing IEJ and SEJ respectively.

3.6.3. Detection of Erroneous Execution

Error detection latency is defined as the time between the initiation of erroneous behaviour and its detection. This parameter is an important performance characteristic when evaluating detection techniques [Blough & Masson, 1987]. The discrete state nature of the microprocessor model presented in this chapter means that error detection latency is determined as a function of the number of erroneously processed instructions during erroneous behaviour.

Detection of erroneous behaviour can be provided by implementation of restart outcomes. Restart outcomes take execution to an address space location predefined by the processor architecture. A recovery routine can be placed at this address. Now restart outcomes produce controlled return to the recovery routine. Hence erroneous behaviour is detected by a restart outcome. In order to remove any ambiguity, recovery routines are only placed for restart outcomes generated by erroneous behaviour.

Detection of erroneous behaviour may occur during the linear erroneous execution following an IEJ, or one or more SEJs. The probability of detection $D(k)$ on the *kth* processed instruction of erroneous execution is given by,

$$D(k) = D_{IEJ}(k) + D_{SEJ}(k), \qquad \text{where } k \geq 0. \tag{3.13.}$$

where $D_{IEJ}(k)$ and $D_{SEJ}(k)$ respectively represent the detection coefficients of erroneous execution following either an IEJ or SEJ.

The detection coefficients are derived using the execution characteristics of erroneous behaviour during processing following an IEJ and SEJ. Let $P_{RT}^{IEJ}(k)$ and $P_{RT}^{SEJ}(k)$ represent the probability of the *kth* instruction processed yielding detection (restart outcome) of erroneous behaviour following an IEJ and SEJ respectively.

The detection coefficient $D_{IEJ}(k)$ is the probability of detecting erroneous execution on the *kth* instruction before a SEJ occurs,

$$D_{IEJ}(k) = P_{RT}^{IEJ}(k), \qquad \text{where } k \geq 0. \tag{3.14.}$$

The detection coefficient $D_{SEJ}(k)$ sums the probability of all possible execution paths resulting in detection after one or more SEJs. Every such execution route requires at least one SEJ, other than a stop/wait or restart outcome representing a processing stall and error detection respectively, after the IEJ commencing erroneous behaviour.

$$D_{SEJ}(k) = \sum_{m=0}^{k} P_{J'}^{IEJ}(m).\left[\sum_{n=0}^{k-m} \Psi(n).P_{RT}^{SEJ}(k-m-n)\right]. \tag{3.15a.}$$

where,

$$\Psi(n) = \begin{cases} 1 & \text{, when } n = 0, \\ \sum_{z=1}^{n} P_{J'}^{SEJ}(z).\Psi(n-z) & \text{, when } n \geq 1. \end{cases} \tag{3.15b.}$$

Equation (3.15b.) calculates the probability associated with periods of erroneous execution initiated by and terminating with an unspecified jump or return outcome.

To demonstrate the function of equation (3.13.) consider a simple example to determine the probability of detecting erroneous execution before four instructions have been erroneously processed. That is, evaluate $D(k)$ when $k = 3$. From equation (3.13.),

$$D(3) = D_{IEJ}(3) + D_{SEJ}(3) \tag{3.16.}$$

Equation (3.14.) yields the probability of detecting erroneous execution when no SEJs occur,

$$D_{IEJ}(3) = P_{RT}^{IEJ}(3) \tag{3.17.}$$

Equation (3.15.) yields the probability of detecting erroneous execution when one or more SEJs occur,

$$
\begin{aligned}
D_{SEJ}(3) = & P_{J'}^{IEJ}(0).1.P_{RT}^{SEJ}(3)+ \\
& P_{J'}^{IEJ}(0).P_{J'}^{SEJ}(1).1.P_{RT}^{SEJ}(2)+ \\
& P_{J'}^{IEJ}(0).P_{J'}^{SEJ}(1).P_{J'}^{SEJ}(1).1.P_{RT}^{SEJ}(1)+ \\
& P_{J'}^{IEJ}(0).P_{J'}^{SEJ}(2).1.P_{RT}^{SEJ}(1)+ \\
& P_{J'}^{IEJ}(0).P_{J'}^{SEJ}(1).P_{J'}^{SEJ}(1).P_{J'}^{SEJ}(1).1.P_{RT}^{SEJ}(0)+ \\
& P_{J'}^{IEJ}(0).P_{J'}^{SEJ}(1).P_{J'}^{SEJ}(2).1.P_{RT}^{SEJ}(0)+ \\
& P_{J'}^{IEJ}(0).P_{J'}^{SEJ}(2).P_{J'}^{SEJ}(1).1.P_{RT}^{SEJ}(0)+ \\
& P_{J'}^{IEJ}(0).P_{J'}^{SEJ}(3).1.P_{RT}^{SEJ}(0)+ \\
& P_{J'}^{IEJ}(1).1.P_{RT}^{SEJ}(2)+ \\
& P_{J'}^{IEJ}(1).P_{J'}^{SEJ}(1).1.P_{RT}^{SEJ}(1)+ \\
& P_{J'}^{IEJ}(1).P_{J'}^{SEJ}(1).P_{J'}^{SEJ}(1).1.P_{RT}^{SEJ}(0)+ \\
& P_{J'}^{IEJ}(1).P_{J'}^{SEJ}(2).1.P_{RT}^{SEJ}(0)+ \\
& P_{J'}^{IEJ}(2).1.P_{RT}^{SEJ}(1)+ \\
& P_{J'}^{IEJ}(2).P_{J'}^{SEJ}(1).1.P_{RT}^{SEJ}(0)+ \\
& P_{J'}^{IEJ}(3).1.P_{RT}^{SEJ}(0),
\end{aligned}
\tag{3.18.}
$$

where equation (3.7.) and equation (3.8.) define $P_{J'}^{IEJ}(0) = 0$, $P_{J'}^{SEJ}(0) = 0$, and $P_{RT}^{SEJ}(0) = 0$,

Substituting equation (3.17.) and equation (3.18.) into equation (3.16.) gives,

$$
\begin{aligned}
D(3) = & P_{RT}^{IEJ}(3)+ \\
& P_{J'}^{IEJ}(1).1.P_{RT}^{SEJ}(2)+ \\
& P_{J'}^{IEJ}(1).P_{J'}^{SEJ}(1).1.P_{RT}^{SEJ}(1)+ \\
& P_{J'}^{IEJ}(2).1.P_{RT}^{SEJ}(1).
\end{aligned}
\tag{3.19.}
$$

This simple example demonstrates the function of equation (3.13.). Although complex, the equation does simplify to produce an almost intuitive result.

Error detection latency L_d can be determined for the microprocessor model by calculating the expectation of detecting erroneous behaviour:

$$L_d = E[K] = \sum_k k.D(k), \tag{3.20.}$$

where K is the random variable of the detection function $D(k)$.
This formula determines the mean number of instructions expected to be processed before detection.

3.6.4. Erroneous Execution Stall

A processing stall is considered to occur when the microprocessor enters a stop/ wait state. Execution stalls require an external hardware interrupt to facilitate state exit and continuing execution, but the occurrence of such events during erroneous execution is unknown. It is therefore important to predict the significance of this eventuality. The following evaluations determine the probability of a processing stall during erroneous execution.

Erroneous execution may stall during the linear erroneous execution following an IEJ, or one or more SEJs. The probability of stalling $S(k)$ on the *kth* processed instruction of erroneous execution is given by,

$$S(k) = S_{IEJ}(k) + S_{SEJ}(k), \qquad \text{where } k \geq 0. \tag{3.21.}$$

where $S_{IEJ}(k)$ and $S_{SEJ}(k)$ respectively represent the stalling coefficients of erroneous execution following either an IEJ or SEJ.

The stalling coefficients are derived using the execution characteristics of erroneous behaviour during processing following an IEJ and SEJ. Let $P_{SW}^{IEJ}(k)$ and $P_{SW}^{SEJ}(k)$ represent the probability of the *kth* instruction processed stalling (stop/wait outcome) erroneous behaviour following an IEJ and SEJ respectively.

The stalling coefficient $S_{IEJ}(k)$ is given by,

$$S_{IEJ}(k) = P_{SW}^{IEJ}(k), \qquad \text{where } k \geq 0. \tag{3.22.}$$

which is the probability of execution following an IEJ terminating with a stall outcome before a SEJ occurs.

The stalling coefficient $S_{SEJ}(k)$ represents all possible execution routes to a stall incorporating one or more SEJs. Every such execution route requires at least one SEJ, other than a stop/wait or restart outcome representing $S_{IEJ}(k)$ and detection respectively, after the IEJ commencing erroneous behaviour ($P_{J'}^{IEJ}$). The stalling execution path may or may not include SEJs propagating further periods of linear erroneous execution (Ψ). Finally, an execution path generating a processing stall must terminate with a stop/wait outcome (P_{SW}^{SEJ}). Hence,

$$S_{SEJ}(k) = \sum_{m=0}^{k} P_{J'}^{IEJ}(m).\left[\sum_{n=0}^{k-m} \Psi(n).P_{SW}^{SEJ}(k-m-n)\right]. \qquad (3.23.)$$

where $\Psi(n)$ is defined by equation (3.15b.).

The recursive nature of equation (3.15.) is similar to the above equation, and its functional description can be shared.

The stalling latency L_s can be determined for the microprocessor model by calculating the expectation of stalling erroneous behaviour:

$$L_s = E[K] = \sum_{k} k.S(k), \qquad (3.24.)$$

where K is the random variable of the stalling function $S(k)$.

3.7. Reliability Analysis

The reliability model proposed here defines a microprocessor system failure as hazardous behaviour rather than loss of function. Hazardous behaviour is unpredictable and may mutilate system integrity and/or lead to catastrophic failure. The model outlined below describes how loss of function may not immediately induce hazardous behaviour if there is automatic repair.

The reliability of a microprocessor system can be analysed using a state/time random variable stochastic model. Let a microprocessor occupy one of two behavioural states: *controlled (C)*, and *uncontrolled (U)*. The processor remains in a controlled

state until the occurrence of an event induces a transition to the uncontrolled state. Such transitions are considered system failures and mark the initiation of hazardous microprocessor operation. The probability of a transition in time δt is $\lambda(t).\delta t$, where $\lambda(t)$ is the failure rate. The reliability model is shown in Figure 3.3. The model is called a Markov Process because of its discrete-state, continuous-time nature.

3.7.1. Failure Rate, λ(t)

Let the sample space E_s, comprise of a set of events corresponding to initiated erroneous behaviour, that is, IEJs. Let $E_r \in E_s$ where E_r is an event leading to recovery, and $E_f \in E_s$ where E_f is an event leading to failure. Within the sample space the conditions $E_r \cup E_f = E_s$, and $E_r \cap E_f = \emptyset$ exist.

Let the probability of the event E_r and E_f be $P(E_r)$ and $P(E_f)$ respectively, which leads to

$$P(E_r) + P(E_f) = 1 \tag{3.25.}$$

Now, assuming the event E_s occurs randomly at a rate of q events per hour, then the failure rate of the microprocessor is given by

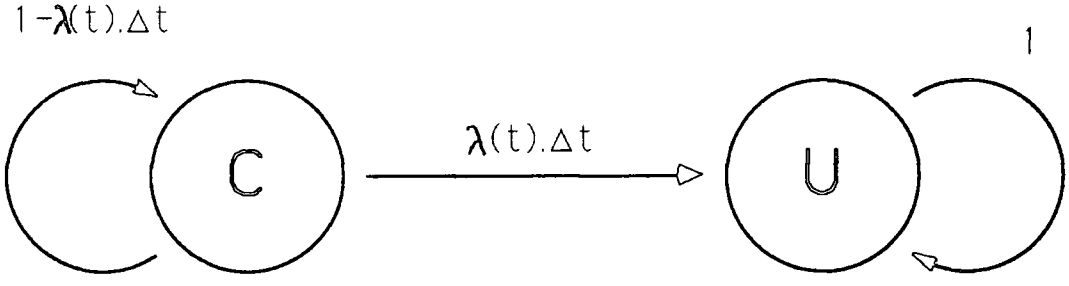
$$\lambda(t) = q.P(E_f) \tag{3.26.}$$

The failure rate is not time dependent giving,

$$\lambda(t) = \text{ constant } = \lambda \tag{3.27.}$$

so the Markov Process is termed homogeneous.

Equation (3.25.) shows that the probability of an event leading to failure is dependent on the event not leading to recovery. The probability of recovery is itself dependent on the detection of erroneous behaviour.



C : Controlled Behaviour State
U : Uncontrolled Behaviour State
$\lambda(t)$: Failure Rate

Figure 3.3. : Reliability Model

3.7.2. Probability of an Event E Leading to Failure, $\mathbf{P(E_f)}$

A stringent specification of failure requires the uncontrolled (erroneous) execution of one or more instructions to be complete. The failure event is therefore any outcome, other than a restart, generated on or before completion of the first erroneously processed instruction following an IEJ. The detection capability of a restart outcome means that its operation is controlled. The probability of a failure event is expressed as,

$$P(E_f) = 1 - \sum_{k}\{D(k)\}, \qquad \text{where } k \leq 1. \tag{3.28.}$$

The cumulative density of the detection function incorporates the probability of detection through the two basic phases of erroneous execution: execution following an IEJ and SEJ. Properties of these phases are now given in respect of the microprocessor model. A jump outcome, other than restart, following an IEJ or SEJ can only occur when an instruction has completed its processing. Equations (3.7a.) and (3.8a.) yield $P_{J'}^{IEJ}(0)$ and $P_{J'}^{SEJ}(0)$ with nil probabilities. Equation (3.8a.) also yields $P_{RT}^{SEJ}(0)$ a nil probability. Evaluating the effective instruction distribution in the address space given by equation (3.7.) yields $P_{RT}^{IEJ}(0) = \beta$, and $P_{RT}^{IEJ}(1) = (1-\beta)[P_{RT}(1) + \gamma.P_{J'}(1)]$. Substituting equation (3.13.) into equation (3.28.) and applying these conditions gives,

$$P(E_f) = 1 - [\beta + (1-\beta)(P_{RT}(1) + \gamma.P_{J'}(1))] \tag{3.29.}$$

substituting equations (3.2.) and (3.6.),

$$P(E_f) = 1 - \left[\beta + (1-\beta)(\frac{Ne_{RT}}{Ne_J}.P_J + \gamma.\left(\frac{Ne_{UJ} + Ne_{RN} + Ne_{S/W}}{Ne_J}\right).P_J)\right], \tag{3.30.}$$

substituting equation (3.4.),

$$P(E_f) = 1 - \left[\beta + \frac{(1-\beta)}{N_L}.\left(Ne_{RT} + \gamma.[Ne_{UJ} + Ne_{RN} + Ne_{S/W}]\right)\right], \tag{3.31.}$$

$$P(E_f) = (1 - \beta)\left(1 - \frac{Ne_{RT} + \gamma[Ne_{UJ} + Ne_{RN} + Ne_{S/W}]}{N_L}\right). \tag{3.32.}$$

and from equation (3.25.),

$$P(E_r) = \left[\beta + \frac{(1-\beta)}{N_L}.\left(Ne_{RT} + \gamma.[Ne_{UJ} + Ne_{RN} + Ne_{S/W}]\right)\right]. \tag{3.33.}$$

3.7.3. Reliability Evaluation

Consider the respective probabilities, for the reliability model, of being in a controlled state or uncontrolled state at time $t + \delta t$.

$$P_c(t + \delta t) = [1 - \lambda(t).\delta t].P_c(t), \tag{3.34.}$$

$$P_u(t + \delta t) = [\lambda(t).\delta t].P_c(t) + 1.P_u(t). \tag{3.35.}$$

Substituting equation (3.27.) and re-arranging equations (3.34.) and (3.35.) gives,

$$\frac{P_c(t + \delta t) - P_c(t)}{\delta t} = -\lambda.P_c(t), \tag{3.36.}$$

$$\frac{P_u(t + \delta t) - P_u(t)}{\delta t} = \lambda.P_c(t), \tag{3.37.}$$

and passing to a limit as $\delta t \rightarrow 0$ yields,

$$\frac{d\{P_c(t)\}}{dt} = -\lambda.P_c(t), \tag{3.38.}$$

$$\frac{d\{P_u(t)\}}{dt} = \lambda.P_c(t). \tag{3.39.}$$

Re-arranging and integrating equation (3.38.),

$$\int \frac{d\{P_c(t)\}}{P_c(t)} = -\lambda \int dt, \tag{3.40.}$$

$$ln\{P_c(t)\} = -\lambda t + C_1, \tag{3.41.}$$

$$P_c(t) = exp\{-\lambda t + C_1\}. \tag{3.42.}$$

Applying the initial conditions; when $t = 0$, then $P_c(t) = 1$ and $P_u(t) = 0$ giving $C_1 = 0$. Hence equation (3.42.) becomes,

$$P_c(t) = exp\{-\lambda t\}. \tag{3.43.}$$

Now re-arranging and integrating equation (3.39.) gives,

$$\int d\{P_u(t)\} = \lambda \int P_c(t).dt, \tag{3.44.}$$

$$\int d\{P_u(t)\} = \lambda \int e^{-\lambda t} dt, \tag{3.45.}$$

$$P_u(t) = exp\{-\lambda t\} + C_2. \tag{3.46.}$$

Again applying the initial conditions; when $t = 0$, then $P_c(t) = 1$ and $P_u(t) = 0$ giving $C_2 = 1$. Hence equation (3.46.) becomes,

$$P_u(t) = 1 - exp\{-\lambda t\}. \tag{3.47.}$$

The reliability of the microprocessor system in the model is given by the probability of the system remaining in a controlled state. That is,

$$R(t) = P_c(t), \tag{3.48.}$$

$$R(t) = exp\{-\lambda t\}, \tag{3.49.}$$

substituting equations (3.26.) and (3.27.) gives

$$R(t) = exp\left\{-qt.(1-\beta).\left(1 - \frac{Ne_{RT} + \gamma[Ne_{UJ} + Ne_{RN} + Ne_{S/W}]}{N_L}\right)\right\}. \tag{3.50.}$$

3.7.4. Mean Time To Failure

The concept of Mean Time To Failure (MTTF), used in hardware reliability calculations, can be adapted for this work. It provides a method of comparing hardware and software reliability.

MTTF is defined as,

$$MTTF = \int_0^\infty R(t).dt, \tag{3.51.}$$

substituting (3.49.) gives,

$$MTTF = \int_0^\infty exp\{-\lambda t\}.dt, \tag{3.52.}$$

$$MTTF = \left[\frac{exp\{-\lambda t\}}{\lambda}\right]_0^\infty, \tag{3.53.}$$

$$MTTF = \frac{1}{\lambda}. \tag{3.54.}$$

Substituting equations (3.26.) and (3.27.) gives,

$$MTTF = \frac{1}{q.P(E_f)}, \tag{3.55.}$$

and substituting equation (3.32.) gives,

$$MTTF = \left[\frac{N_L}{q(1-\beta)(N_L - Ne_{RT} - \gamma.[Ne_{UJ} + Ne_{RN} + Ne_{S/W}]}\right]. \tag{3.56.}$$

3.8. Availability

The availability of a microprocessor is the proportion of time for which the microprocessor is fully operational. An inherent assumption made when calculating availability is that the target system is maintained, i.e. the system has its operation restored after failure. Within the microprocessor model presented in this chapter, restoration is provided by the automatic execution of a recovery routine when a restart outcome is generated during erroneous execution. Availability A_v is dependent on Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR),

$$A_v = \left[\frac{MTTF}{MTTF + MTTR}\right]. \tag{3.57.}$$

The Mean Time To Failure (MTTF) is defined by equation (3.54.). The Mean Time To Repair (MTTR) includes all processing before the microprocessor is restored to its fully operational state. In order to facilitate repair the microprocessor model must allow for the implementation of a recovery routine. Mean Time To Repair can be estimated using the following equation.

$$MTTR = \left(\frac{L_d + N_R}{I_f}\right), \tag{3.58.}$$

where L_d is the mean number of instructions processes erroneously before detection of erroneous behaviour (error latency from equation 3.20.), N_R is the mean number of instructions executed after detection by the recovery routine, and I_f is the mean number of instructions executed per hour.

Substituting equations (3.54.) and (3.58.) into (3.57.) gives an estimate for the availability of a microprocessor system that employs coverage for the erroneous behaviour described in this chapter.

$$A_v = \left[\frac{\frac{1}{\lambda}}{\frac{1}{\lambda} + \left(\frac{L_d + N_R}{I_f}\right)}\right], \tag{3.59.}$$

$$A_v = \left(\frac{1}{1 + \frac{\lambda}{I_f}.[L_d + N_R]}\right). \tag{3.60.}$$

3.9. Summary

The event, induced by a temporary fault, initiating erroneous microprocessor behaviour is defined as an Initial Erroneous Jump (IEJ). Erroneous behaviour is characterized by periods of linear erroneous execution interspersed by erroneous jumps. The characteristics of erroneous execution following an IEJ or SEJ can be statistically modelled. Error latency is derived from detection capabilities in the microprocessor model. Failure mode analysis is used within a Markov Model to determine functions of reliability and Mean Time To Failure (MTTF). Availability of the microprocessor system by the model is estimated under the assumption that a recovery routine is implemented. These functions allow the comparative assessment of recovery techniques to be made for software disrupted by temporary faults in a form which can be related to calculations for permanent faults in digital systems.

Chapter 4

EVALUATING MICROPROCESSOR BEHAVIOUR

CHAPTER FOUR

EVALUATING MICROPROCESSOR BEHAVIOUR

4.1. Introduction

A model of erroneous microprocessor behaviour is presented in the previous chapter. This chapter applies the model to a selection of target processors which include 8, 16, and 32-bit architectures using instruction mix analysis.

The chapter commences with the derivation of parameter values required by the model for the microprocessors under investigation. The content of the address space is assumed to be random for the purpose of statistical analysis. Characteristics of erroneous execution are described for each of the target processors. In particular the possibilities of catastrophic failure and recovery are investigated because of their influence on the dependability of a microprocessor based system.

Finally, the reliability of a microprocessor system is considered. A comparison is made between the microprocessors modelled using the reliability parameter Mean Time To Failure (MTTF). Reliability calculations assume that the host processor has no recovery capability. Many of the microprocessors investigated, however, do have a recovery capability provided by the detection attribute of instructions that develop a restart outcome. In order to assess the performance of such maintained microprocessor systems, the microprocessor systems availability is evaluated.

4.2. Instruction Mix Analysis

The model of erroneous microprocessor behaviour presented in Chapter 3 is evaluated using instruction mix analysis. This involves determining the mean instruction distribution for a section of memory and modelling the expected behavioural characteristics. Such modelling is abstracted from actual microprocessor behaviour which is dependent on instruction sequences. Nevertheless, instruction mix analysis does provide a valuable method for indicating the nature of erroneous microprocessor behaviour and its variation between target processors.

This chapter evaluates the erroneous behaviour of a selection of microprocessors. The processors considered are: Motorola 6800, Intel 8048, Intel 8085, Intel 8086, Motorola 68000, Motorola 68010, AMD Am29000, Motorola 68020, and Intel 80386. The microprocessors are chosen to include common application examples of 8, 16, and 32-bit architectures. In addition, these processors implement various design features including reduced instruction sets, ROM instruction decoders, and instruction processing exceptions.

The instruction mix of the target processors is shown in Table 4.1., data being collated from Appendix A. Each instruction set is divided into instruction state outcomes, *non-jump*, *restart*, *undefined jump*, *return*, and *stop/wait*. The instruction mix of the undefined instructions within the processor instruction sets is detailed in Table 4.2. Some of these instruction sets contain unspecified instructions which through experiment have been defined [Halse, 1984].

4.3. Architecture Parameters For The Microprocessor Model

This section determines the parameter values for the model of erroneous microprocessor behaviour which are dependent on the processor architecture.

4.3.1. Built-In Microprocessor Detection Capability

The Motorola 68000 family of microprocessors execute instruction op-codes residing at an even byte boundary location in the address space. If an attempt to process an instruction op-code at an odd byte boundary location in the address space is made, then an immediate 'restart' outcome is entered. The outcome is assumed to be immediate because the 'odd byte address' exception does not process the instruction op-code concerned but rather after a few clock cycles determines an illegal odd address has been accessed. The β parameter defined in equations (3.7.) and (3.8.) will therefore have an inherent value of 0.5 for this family of microprocessors.

The remaining microprocessors evaluated within this chapter do not have this or a similar method of generating a restart outcome during instruction processing. The models of these microprocessors therefore define β to be zero.

Instruction Type	Microprocessor								
	8-Bit			16-Bit			32-Bit		
	MC 6800	Intel 8048	Intel 8085	Intel 8086	MC 68000	MC 68010	AMD 29000	MC 68020	Intel 80386
Non-Jump	173	183	209	262	39039	39217	151	41623	333
Stop/Wait	1	0	1	2	1	1	1	1	1
Return	2	2	5(4)	5	3	4	2	4	5
Unspecified Jump	20(14)	45(20)	19(16)	38(20)	4280(4278)	4280(4278)	12(7)	4792(4790)	49(37)
Restart	1	0	11(4)	2	19	19	20(20)	175(104)	3
Undefined	59	26	10	31	22194	22015	70	18941	225
TOTAL	256	256	256	340	65536	65536	256	65536	616

Note : Data from Appendix A. Brackets denote the number of those instructions which are dependent on a condition for their operation.

Table 4.1. : Microprocessor Instruction Set Evaluation

Undefined Instruction Type	Microprocessor								
	8-Bit			16-Bit			32-Bit		
	MC 6800	Intel 8048	Intel 8085	Intel 8086	MC 68000	MC 68010	AMD Am29000	MC 68020	Intel 80386
Non-Jump	49	22	7	25	0	0	0	0	0
Stop/Wait	4	0	0	0	0	0	0	0	0
Return	2	0	0	4	0	0	0	0	0
Unspecified Jump	4(1)	4(3)	2(2)	2	0	0	0	0	0
Restart	0	0	1(1)	0	22194	22015	70	18941	225
TOTAL	59	26	10	31	22194	22015	70	18941	225

Note : Data from Appendix A. Brackets denote the number of those instructions which are dependent on a condition for their operation.

Table 4.2. : Microprocessor Undefined Instruction Evaluation

4.3.2. Modelling the Microprocessor Program Counter

Most of the microprocessors evaluated in this chapter have a single program counter which is capable of specifying every location in the address space. The Intel 8086 and 80386 microprocessors, however, have an internal address bus smaller than its external address bus. To derive a location in the 8086 address space it uses two *program counters*. The address put on the external address bus is the sum of the 16-bit Instruction Pointer and the 16-bit Control Segment Register which has already been left shifted four bits. Hence a 20-bit location is put on the external address bus. This method of deriving the address bus value means that corruption of either the Instruction Pointer or the Control Segment Register corrupts the microprocessor's effective *program counter*. The model considers the generation of an erroneous jump as corruption of the single effective program counter.

4.3.3. Instruction Processing Exceptions

Some microprocessor instruction sets include instructions which generate a restart outcome when an abnormal processing condition is identified. A good example common to most microprocessors is the 'divide by zero' processing exception. Processing exceptions should not be confused with conditional instruction outcomes where a test is incorporated into the instruction operation in order to determine whether or not a task is performed, e.g. conditional branch. For the purpose of statistical analysis within this chapter, instruction processing exceptions are considered not to occur. The γ parameter used in equation (3.7.) and (3.8.) is therefore zero.

4.4. Evaluating Microprocessor Models of Erroneous Behaviour

Erroneous execution within the used area is initially modelled by execution through a memory of random content. The two main behavioural characteristics, linear erroneous execution and erroneous jumps, are investigated.

A selection of microprocessor instruction sets are examined in Table 4.3. The distribution of instructions through the address space of random content for many target processors is the same as their instruction set mix. In many instances, this is

Instruction Type	Microprocessor								
	8-Bit			16-Bit			32-Bit		
	6800	8048	8085	8086	68000	68010	29000	68020	80386
Non-Jump	89.648%	84.570%	89.648%	83.203%	62.823%	63.104%	64.258%	67.245%	86.121%
Stop/Wait	1.953%	0.000%	0.391%	0.781%	0.002%	0.002%	0.391%	0.002%	0.391%
Return	1.563%	0.782%	1.172%	3.516%	0.005%	0.006%	0.782%	0.006%	1.953%
Unspecified Jump	6.445%	14.648%	4.688%	11.719%	3.267%	3.267%	3.320%	3.658%	9.961%
Restart	0.391%	0.000%	3.711%	0.781%	0.029%	0.029%	3.906%	0.187%	7.324%
Undefined	*	*	*	*	33.865%	33.592%	27.343%	28.902%	2.147%

Notes : Instruction set data from Appendix A. The instruction set distributions for the Intel 8086 and 80386 microprocessors have been manipulated for random data interpretation. The symbol '*' denotes that any undefined instructions have been grouped with their associated instruction type. Undefined instructions without the symbol '*' are specified as having a restart outcome. Conditional jump instructions are evaluated as having an equal chance of activation or non-activation, reference equations (3.3.) and (3.4.).

Table 4.3. : Random Data Instruction Interpretation

due to the processor implementing a ROM instruction decoder, within its architecture, which specifies an instruction for all possible opcode bit formats. The Intel 8086 and 80386, however, require their instruction set mix to be manipulated to reflect a random data instruction mix.

Linear erroneous execution state outcomes are plotted as cumulative functions using equation (3.9.) in Figure 4.1. (8-bit processors), Figure 4.2. (16-bit processors), and Figure 4.3. (32-bit processors). The cumulative probability that linear erroneous execution has a particular state outcome after processing a number of instructions, indicated by the instruction index, is shown by the vertical width of the labelled area at that point. The features of each plot to notice are:

i) *Continued Linear Erroneous Execution*

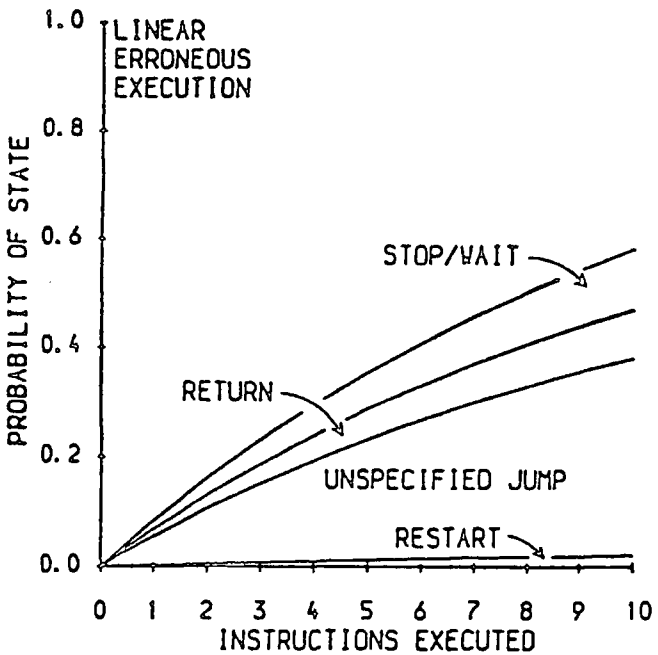
This is the vertical width of the area labelled 'linear erroneous execution' at each instruction index. The larger the vertical width, the greater the probability that linear erroneous execution has continued through the number of instructions indicated by the instruction index.

ii) *Termination by Erroneous Jump*

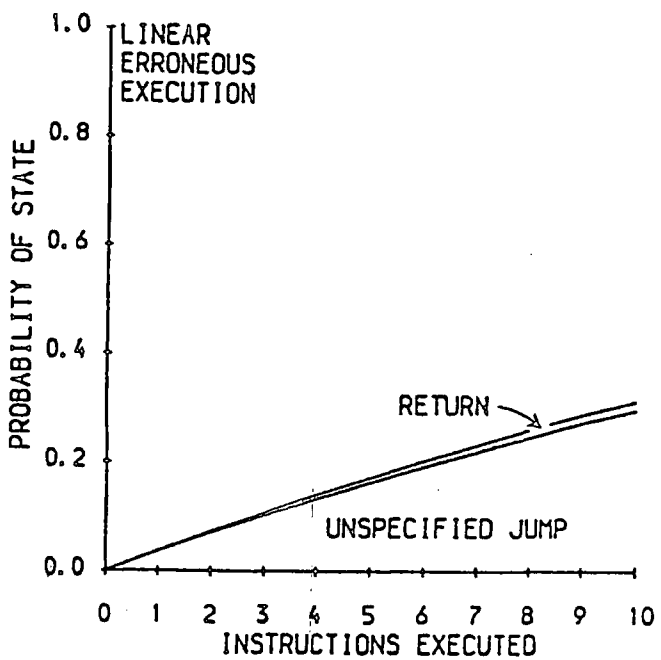
This is the vertical distance of the combined areas beneath the area labelled 'linear erroneous execution' at each instruction index. The larger the vertical width, the greater the probability that linear erroneous execution has been terminated by the present (or any preceding) processed instruction indicated by the instruction index.

iii) *Stop/Wait Outcome*

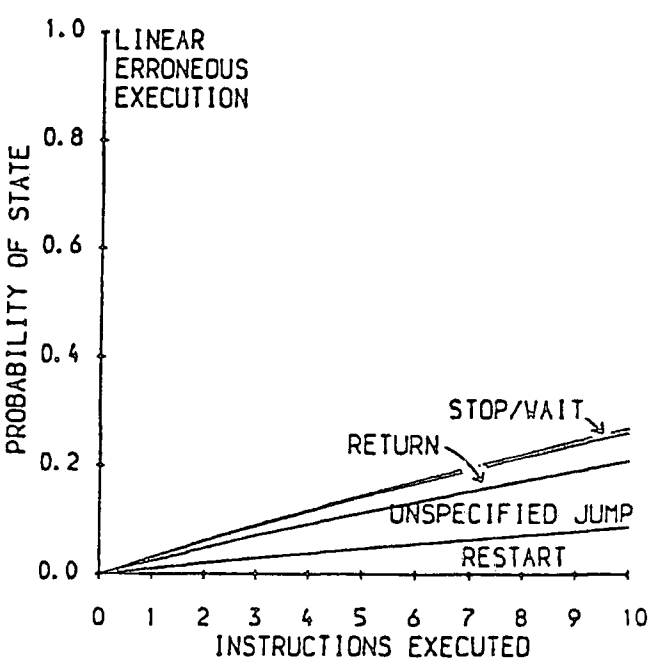
This is the vertical width of the area labelled 'stop/wait' at each instruction index. This outcome represents a catastrophic failure involving the termination of processor activity until the appropriate external interrupt re-initiates operation. The likelihood



(a) MC 6800 Processor

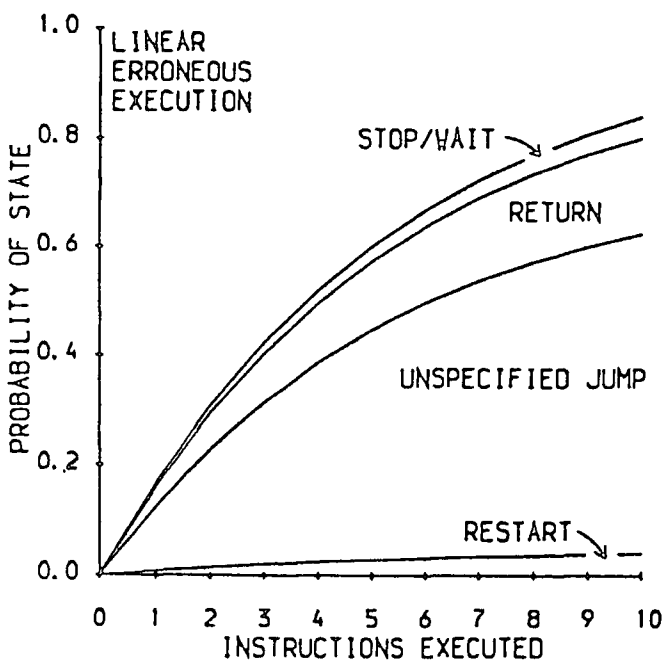


(b) Intel 8048 Processor

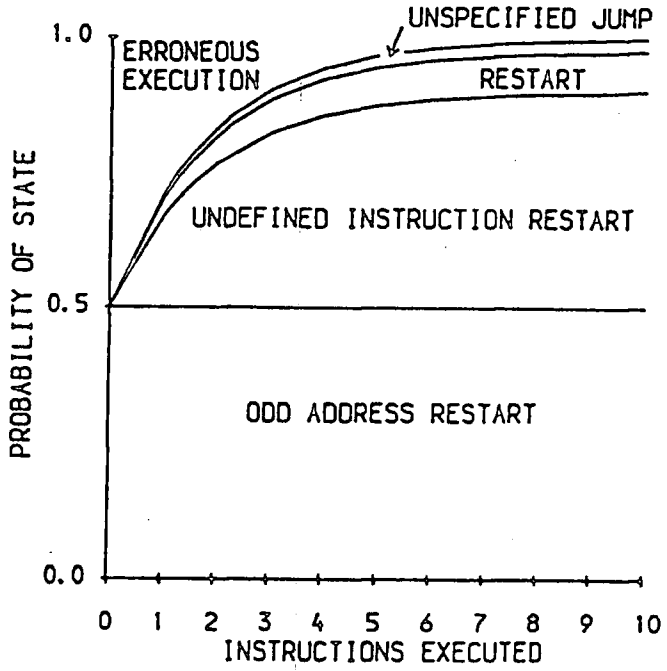


(c) Intel 8085 Processor

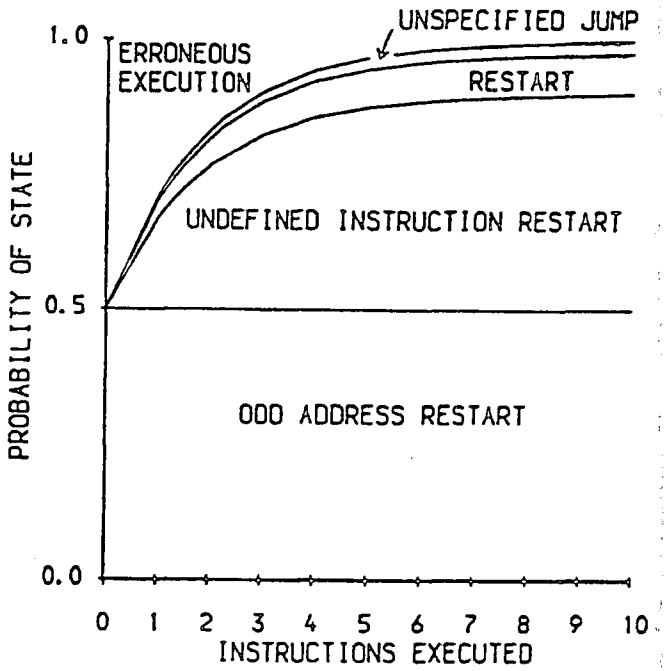
Figure 4.1. : 8-Bit Microprocessor Linear Erroneous Execution



(a) Intel 8086 Processor

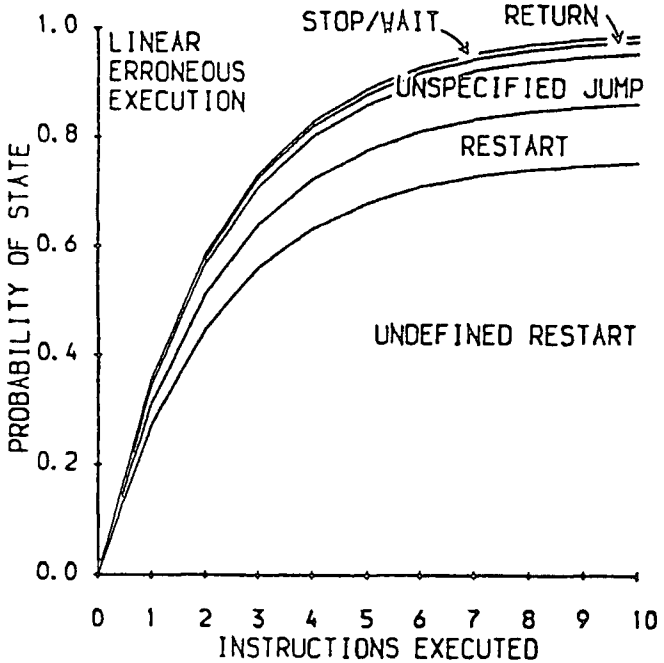


(b) MC 68000 Processor

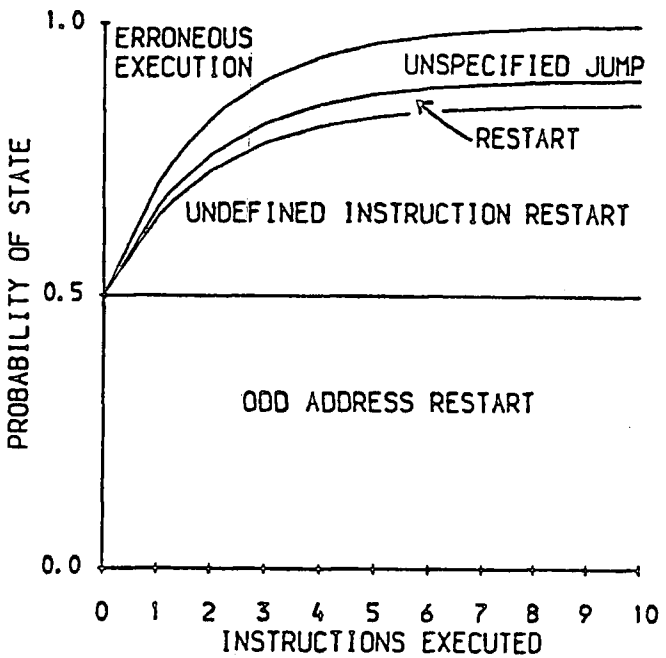


(c) MC 68010 Processor

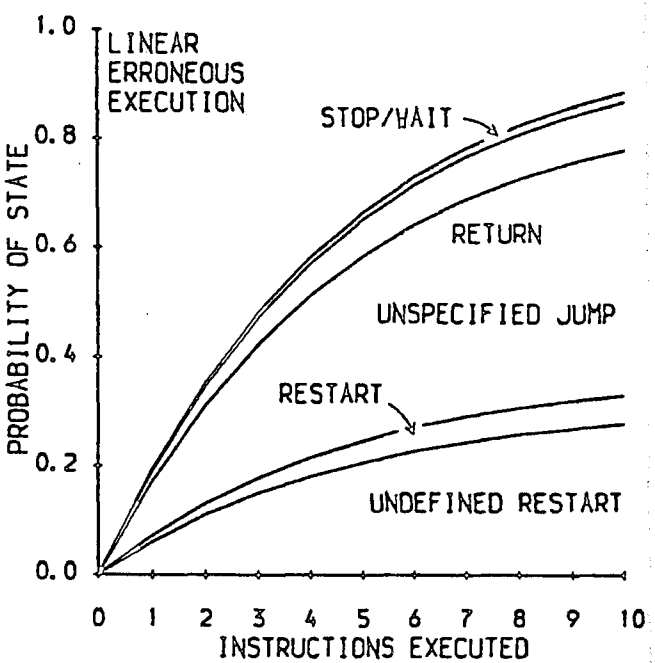
Figure 4.2. : 16-Bit Microprocessor Linear Erroneous Execution



(a) AMD 29000 Processor



(b) MC 68020 Processor



(c) Intel 80386 Processor

Figure 4.3. : 32-Bit Microprocessor Linear Erroneous Execution

of such a restoring event occurring is unpredictable. The larger the vertical width of this area, the higher the probability of this outcome.

iv) *Restart Outcome*

This is the vertical width of the area labelled 'restart' at each instruction index. The restart outcome is the only one that when executed erroneously is considered to generate a controlled outcome. It is for this reason that it will be used for detecting erroneous execution. Hence the 'inherent' detection capability of a microprocessor may be observed by thickness of the 'restart' area. The larger the vertical width, the greater the probability that linear erroneous execution has been detected, and hence terminated, by the present (or any preceding) processed instruction indicated by the instruction index.

The investigation of linear erroneous execution and erroneous jumps gives an indication of the character and attributes of erroneous behaviour.

4.4.1. 8-Bit Processor Evaluations

All the 8-bit microprocessors evaluated exhibit a high probability of periods of linear erroneous execution exceeding ten instructions, see Figure 4.1. In particular the Intel 8048 and 8085 processor models suggest 31% and 27%, respectively, of the periods of linear erroneous execution are expected to terminate within ten instructions. The Motorola 6800 is about twice as likely to terminate a period of linear erroneous execution within ten instructions.

A SEJ terminates all periods of linear erroneous execution in the 8048 microprocessor system. However, for the other two 8-bit processors some periods of linear erroneous execution are terminated by recovery or catastrophic failure. Approximately 79% of the periods of linear erroneous execution are terminated with a SEJ for the Motorola 6800 processor, a similar value of 65% is modelled for the Intel 8085 microprocessor. Although the Intel 8048 processor will never catastrophically fail in

the model, it will also never inherently recover. Within the Motorola 6800 processor non-SEJ terminations of linear erroneous execution as catastrophic failure are expected to occur four more times than recovery. The Intel 8085 processor has a converse relationship, recovery being expected to occur six more times than catastrophic failure.

In summary, the model for the Motorola 6800 microprocessor suggests periods of linear erroneous execution of approximately ten instructions which are approximately five times more likely to terminate in an SEJ than failure, and the chance of recovery is small. The Intel 8048 processor model predicts much longer periods of linear erroneous execution which will always terminate with an SEJ, no catastrophic failure of recovery is possible. Although the Intel 8048 processor will never catastrophically fail in the model, failure is implied by the fact that erroneous execution never ceases. Within the model for the Intel 8085 microprocessor periods of linear erroneous execution are expected of a similar length to those evaluated for the Intel 8048 processor, of which approximately one third terminations are expected to generate recovery, the vast majority of the remaining terminations producing a SEJ.

4.4.2. 16-Bit Processor Evaluations

The instruction mix analysis of erroneous execution presented for the 16-bit microprocessors in Figure 4.2. suggests that these processors have shorter periods of linear erroneous execution than those modelled for the 8-bit processors. The model predicts that in excess of 80% of linear erroneous execution periods will terminate before their tenth processed instruction. The Intel 8086 processor has a mean expected period of linear erroneous execution longer than that for the Motorola 68000 and 68010 microprocessors. This is due to the influence of the instruction set and architecture. The 68000 detection capability is considerably influenced by the 'odd address exception' processor facility. This exception yields a restart outcome for any access to an instruction located at an odd byte address in the memory map.

Within the Intel 8086 processor model there is a 90% probability that a period of linear erroneous execution is terminated by a SEJ. This is much larger than that for the Motorola 68000 and 68010 processors whose model suggests the likelihood of

the same outcome as less than 3%. Hence, not only are the periods of linear erroneous execution expected to be shorter for the Motorola processors than the Intel, but also the Motorola processors will have fewer periods of linear erroneous execution before either catastrophic failure or recovery is attained. The Intel 8086 processor model has a similar probability of linear erroneous execution being terminated by recovery through a restart outcome or catastrophic failure through a stop/wait outcome. The Motorola 68000 processors yields very different results. The probability of a stop/ wait outcome for the Motorola 68000 processors is too small to be shown on Figure 4.2(b & c) whilst the probability of a restart outcome and hence recovery is approximately 93% after just four processed instructions during linear erroneous execution.

4.4.3. 32-Bit Processor Evaluations

The result of model application for a selection of 32-bit processors is shown in Figure 4.3. The microprocessors evaluated are the Advanced Micro Devices Am29000 (Version D), the Motorola 68020, and the Intel 80386.

The influence in the model of the 'odd byte address' exception of the Motorola 68020 microprocessor is clearly seen as the 50% intercept in Figure 4.3(b). This feature greatly reduces the mean expected period of linear erroneous execution as previously described for the Motorola 68000 and 68010 processors. The Advanced Micro Devices Am29000 achieves similar periods of erroneous execution without this architectural feature. Its performance relies totally on its instruction set attributes. Both the AMD Am29000 and Motorola 68020 models predict approximately 90% of linear execution periods of five instructions to terminate. This characteristic is not shared by the Intel 80386 processor model in which approximately 63% of linear erroneous execution periods of five instructions are expected to terminate.

Termination of linear erroneous execution for the AMD Am29000 and Motorola 68020 processor models have about a 90% expectation of of generating a restart outcome and hence detection and recovery. The Intel 80386 does not compare so favourably with a 37% chance of a restart terminating linear erroneous execution. The

Intel 80386 restart probability is, however, higher than all 8-bit and 16-bit processors considered earlier with the exception of the Motorola 68000 microprocessor family.

The probability for the Motorola 68020, as with other members of the Motorola 68000 processor family considered earlier, of catastrophic failure through a stop/wait outcome is too small to be shown in Figure 4.3(b). At 0.0015% it is very small when compared with the 8 and 16-bit processor evaluations. Although the AMD Am29000 and Intel 80386 processors have a larger probability of 0.391% of a stop/wait outcome which is visible in Figure 4.3(a & c), it is still small in relation to other processor model evaluations.

4.5. Catastrophic Failure Analysis

The probability of a stop/wait outcome is identified as a catastrophic failure. Such a processing outcome stalls execution until an external interrupt generates a restart outcome and hence initiates recovery. The occurrence of a stop/wait outcome during periods of linear erroneous execution has been investigated for a selection of 8, 16, and 32-bit microprocessors. It is valuable to further consider the probability of catastrophic failure as a function of general erroneous execution. Figure 4.4. shows the predicted chance of catastrophic failure using instruction mix analysis for a selection of processors. The graph is developed using equation (3.21) and data from Table 4.3.

The Motorola 6800 microprocessor has a significantly higher probability of producing a catastrophic failure than the other evaluated processors. The Intel 8086 has an 8% probability of catastrophic failure after ten instructions have been processed during erroneous execution which is just under half that expected for the Motorola 6800. The remaining processors are plotted as two groups in Figure 4.4. One group, comprising of the Intel 8085, AMD Am29000, and Intel 80386, has twice the expectation of catastrophic failure of the other group of the Motorola 68000 family processors. This can be illustrated by a comparison between the Intel 8085 and AMD Am29000 which have the respective likelihoods of 4% and 2% for catastrophic failure before ten instructions are erroneously processed.

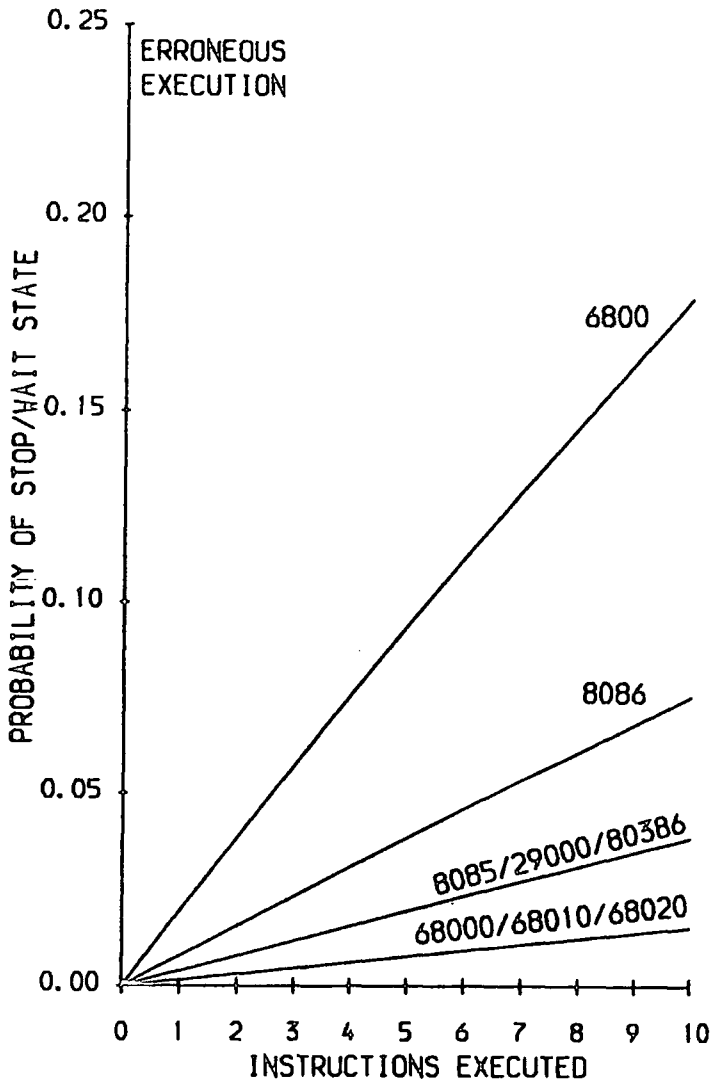


Figure 4.4. : Catastrophic Failure - Instruction Mix Analysis

4.6. Recovery Through The Detection of Erroneous Execution

The probability of detecting erroneous execution is dependent on the generation of a restart outcome. This can be achieved through either an instruction's natural outcome, or a hardware induced outcome where a processing exception occurs. Although the probability of a restart outcome has been considered during periods of linear erroneous execution, it is valuable to evaluate the general function of this outcome during erroneous behaviour. Figure 4.5. shows the probability of a restart outcome during erroneous execution for a selection of processors. The graph is plotted using equation (3.13) and data from Table 4.3.

Both the Motorola 6800 and Intel 8086 have very low probabilities of generating a restart outcome, the former having less than half the expectation of the other. The Intel 8085 shows a significant improvement with a 32% chance of initiated recovery after ten instruction are erroneously processed. This represents over a four-fold improvement on the Intel 8086 processor. The Intel 80386 also shows an enhanced performance with approximately 63% probability that erroneous execution is detected after ten processed instructions. The remaining processors, the AMD Am29000 and Motorola 68000 processor family, have a much better performance. Their instruction mix models suggest the likelihood of erroneous execution being detected after ten erroneously processed instructions is in excess of 97%. The 'odd byte address' exception is a major contributory factor for the performance of the Motorola 68000 processor family. This influence of the exception is shown as the 50% intercept value for the Motorola 68000 processor plots. The performance of the AMD Am29000 processor is dependent solely on its instruction set mix.

4.7. Evaluating Microprocessor Reliability

Microprocessor reliability is calculated under the 'worst case' assumption that any processor activity other than immediate detection is considered as system failure. Detection is provided by those instructions which generate a restart outcome. Immediate detection requires a restart outcome to be attained before the second instruction of erroneous behaviour is processed.

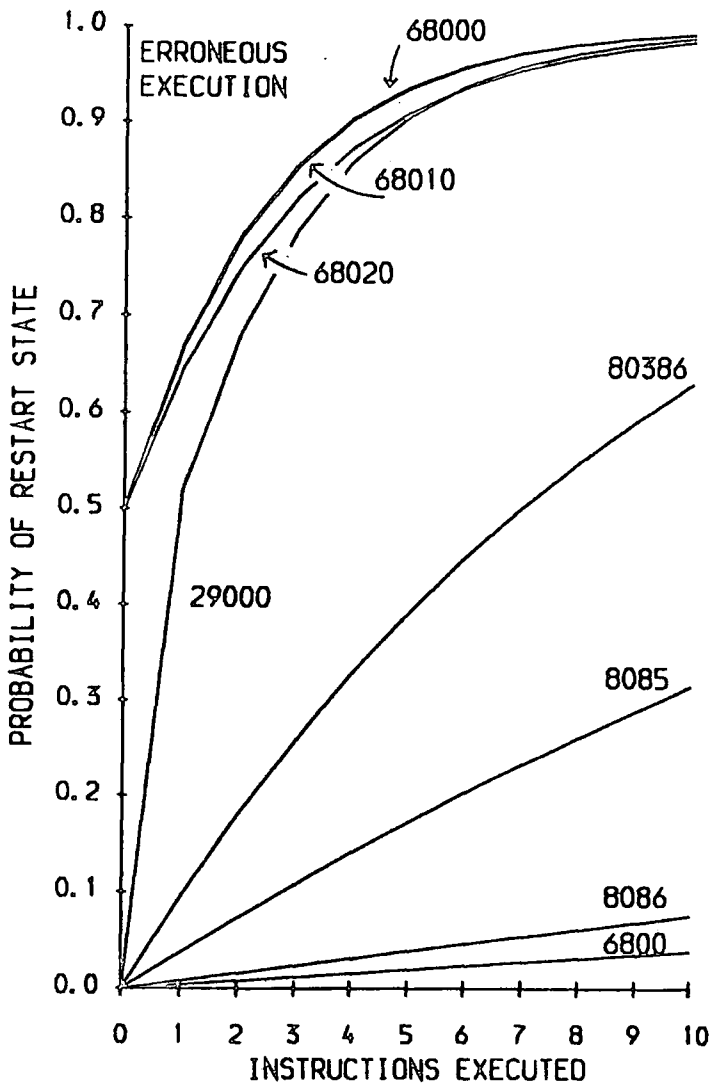


Figure 4.5. : Recovery Through Detection - Instruction Mix Analysis

The reliability of the microprocessors modelled, $R(t)$, is assessed using equations (3.49.) with the substitution of equations (3.27.), (3.26.), and (3.32.), to yield,

$$R(t) = exp\left\{-qt.(1-\beta).\left(1 - \frac{Ne_{RT} + \gamma[Ne_{UJ} + Ne_{RN} + Ne_{SW}]}{N_L}\right)\right\}, \quad (4.1.)$$

where q is the event rate initiating erroneous behaviour, β is the probability of hardware detection of erroneous execution, γ is the probability of a processing exception, Ne_{RT}, Ne_{UJ}, Ne_{RN}, and Ne_{SW} are the effective numbers of restart, unspecified jump, return and stop/wait outcome generating instructions within the instruction mix, and finally, N_L is the number of instructions in the instruction mix.

Details of the instruction mix for reliability evaluation are shown in Table 4.3. The parameter γ is defined in section 4.3.3., for the purpose of statistical analysis, to be zero so equation (4.1.) becomes,

$$R(t) = exp\left\{-qt.(1-\beta).\left(1 - \frac{Ne_{RT}}{N_L}\right)\right\}. \quad (4.2.)$$

The parameter β is set to zero except for the Motorola 68000 microprocessor family where it is set to 50% to represent the 'odd byte address' exception facility. The determination of processor parameters is discussed in section 4.3.

The reliability curves evaluated for the selection of 8, 16, and 32-bit processors considered in this chapter are shown in Figure 4.6. Unfortunately some processor evaluations are so similar that their individual reliability curves cannot be distinguished. In particular this occurs for the Motorola 68000 and 68010 processors, and the Motorola 6800, Intel 8048, and Intel 8086 processors. The variation in the respective processor reliability for these clusters of curves is indicated by the Mean Time To Failure (MTTF) calculations shown in Table 4.4. These calculations assume the events which initiate erroneous behaviour to occur once per month or every 714 hours.

As discussed earlier the Intel 8048 has no detection capability for erroneous execution and hence its MTTF is equivalent to the event rate. The remaining microprocessors have some degree of detection capability depending on instruction mix

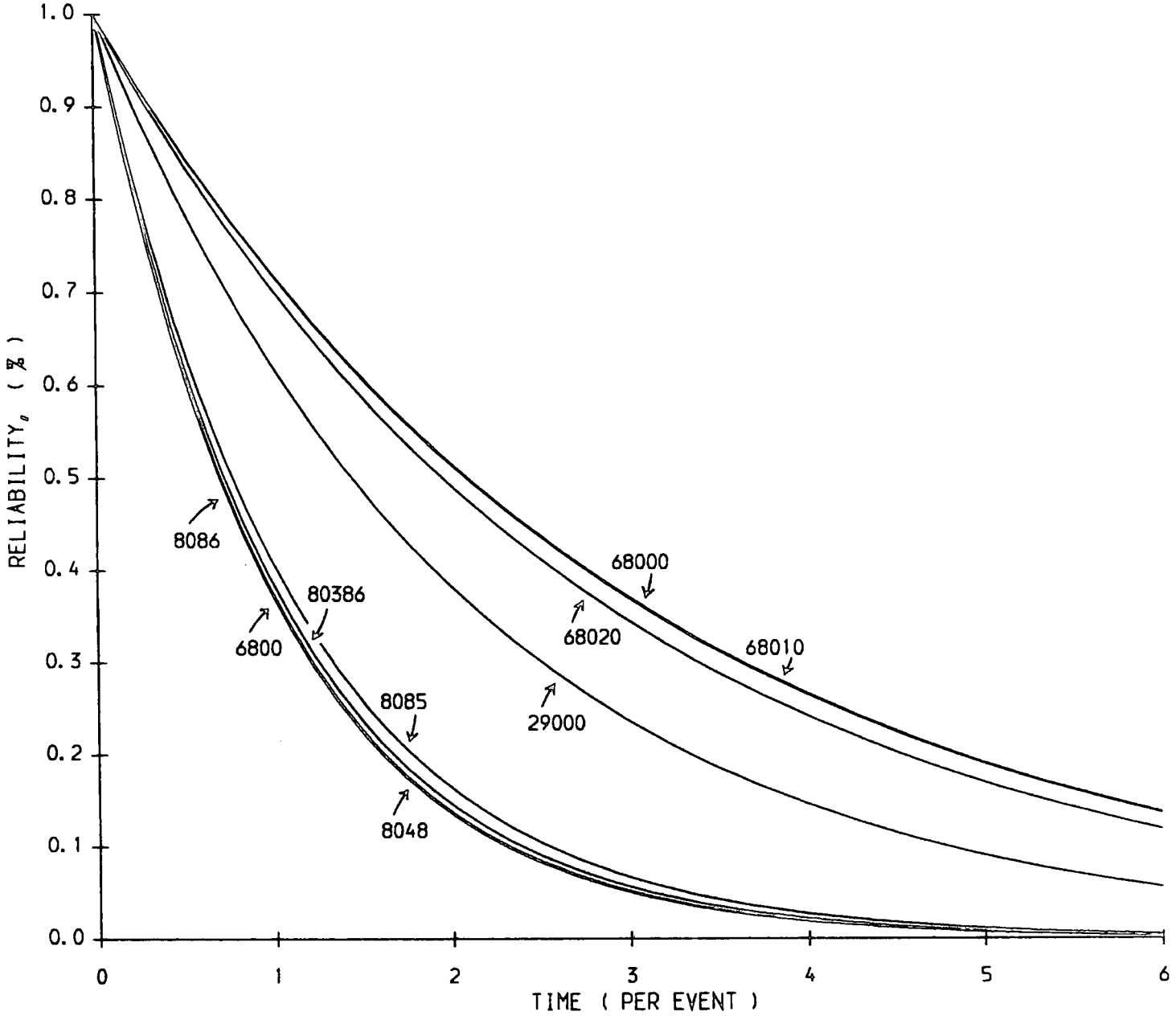


Figure 4.6. : Microprocessor Reliability - Instruction Mix Analysis

Microprocessor		Evaluation Parameters			Mean Time To Failure
Type	Class	γ	β	$P(E_f)$	MTTF
6800	8-Bit	0.0	0.0	0.99609	717 hrs
8048	8-Bit	0.0	0.0	1.00000	714 hrs
8085	8-Bit	0.0	0.0	0.96289	742 hrs
8086	16-Bit	0.0	0.0	0.99219	720 hrs
68000	16-Bit	0.0	0.5	0.33053	2160 hrs
68010	16-Bit	0.0	0.5	0.33190	2151 hrs
29000	16-Bit	0.0	0.0	0.48054	1486 hrs
68020	16-Bit	0.0	0.5	0.35456	2014 hrs
80386	16-Bit	0.0	0.0	0.90529	789 hrs

Notes : MTTF derivation from equation (3.49.) and data from Table 4.3. Transient events initiating erroneous microprocessor behaviour are assumed to occur once per month (every 714 hours).

Table 4.4. : Microprocessor 'Mean Time To Failure' Evaluation

and architectural influences. This detection capability has little effect of the MTTF for the Motorola 6800, Intel 8086, Intel 8085, and Intel 80386 processors. A more significant improvement is shown by the AMD Am29000 processor which has an MTTF approximately double the inter-arrival event period. Finally, the Motorola 68000 processors have the best MTTF which are about three times the inter-arrival event rate. The performance of the Motorola processors can be largely attributed to their 'odd byte address' exception facility which will, under the model conditions, immediately detect half the initiated periods of erroneous microprocessor behaviour.

4.8. Evaluating Microprocessor Availability

The availability of a microprocessor system can only be evaluated if the system is maintained, i.e. manual or automatic repair is facilitated. The model presented for the erroneous behaviour of a microprocessor in Chapter 3 assumes that restart outcome generating instructions can be used to provide a detection capability for erroneous execution. Hence equation (3.60.) can be used to calculate availability,

$$A_v = \left(\frac{1}{1 + \frac{\lambda}{I_f} \cdot [L_d + N_R]} \right). \tag{4.3.}$$

where λ is the rate of failure events (per hour) that initiate erroneous behaviour, I_f is the mean number of instructions executed per hour by the processor, L_d is the error detection latency, and N_R is the number of instructions to be processed in the recovery routine.

A three dimensional availability plot shown in Figure 4.7. describes how availability varies with different event rates and error detection latencies. The solid plane denotes the use of a recovery routine with 50 instructions, whilst the dashed plane shows the effect of a larger recovery routine of 1000 instructions.

In order to aid comprehension of the equation (4.3.) plot consider the following example. A microprocessor operates at 0.4 MHz and has a mean instruction processing time of 400 cycles. Failure events initiating erroneous microprocessor behaviour are expected to occur once per second. Hence,

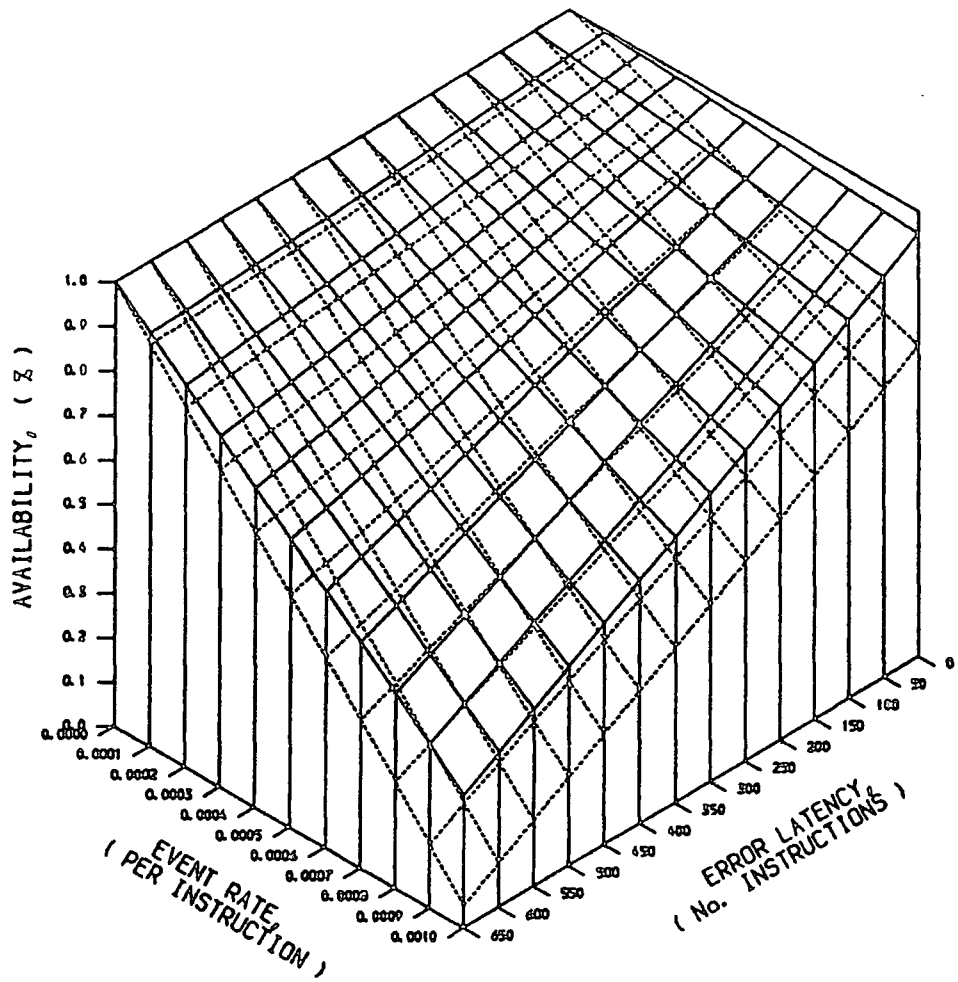


Figure 4.7. : Microprocessor Availability - Instruction Mix Analysis

$$\frac{\lambda}{I_f} = \frac{1.400}{400000} = 0.001 \tag{4.4.}$$

The mean error detection latency (L_d) is ten instructions, and the recovery routine consists of 50 instructions (N_R). The availability of the system can now be determined using equation (4.3.):

$$A_v = \left(\frac{1}{1 + 0.001[10 + 50]} \right), \tag{4.5.}$$

$$A_v = 94.3\%. \tag{4.6.}$$

The calculated availability can be located on Figure 4.7. by mapping the event rate (per instruction) $\frac{\lambda}{I_f}$, and error latency (number of instructions) L_d on the solid surface representing 50 instructions in the recovery routine N_R. The actual operating frequency of a processor may be higher and the mean instruction cycle time shorter than that used in this example. These parameter values would increase the calculated value of system availability.

4.9. Conclusion

A model based on probability theory is used to predict microprocessor erroneous behaviour. Characteristics of linear erroneous execution and their termination are compared for a selection of 8, 16, and 32-bit microprocessors using instruction mix analysis. Whilst this method of analysis does not reflect the time dependency of erroneous behaviour on the instruction sequence, it does provide a valuable insight into the patterns of erroneous behaviour.

Two important processing outcomes are studied: the probability of catastrophic failure and recovery. Catastrophic failure is defined as the entry into a stop/wait state where normal instruction activity ceases. This state can only be exited by the generation of an external interrupt. Such a reset signal cannot be relied upon because they have not been incorporated into the microprocessor system design. The

other important processing outcome is recovery. This is attained through entry into a restart state which by its definition establishes controlled microprocessor activity.

The evaluation of recovery leads to the determination of reliability for the microprocessor system. The selection of target processors is compared using the reliability parameter Mean Time to Failure (MTTF). Many processors show little improvement in the MTTF with respect to the inter-arrival event period. The best results are obtained from the microprocessor models for the AMD Am29000 and Motorola 68000 family which have a three-fold increase in their MTTF compared with the inter-arrival event period.

Finally, the chapter concludes with an investigation of microprocessor system availability. A general plot is shown to indicate the availability of a processor system implementing recovery routine of two sizes and an example discussed. Availability calculations are only made possible where a microprocessor system has an automatic detection and recovery capability for erroneous behaviour.

Chapter 5

DETECTING ERRONEOUS MICROPROCESSOR BEHAVIOUR

CHAPTER FIVE

DETECTING ERRONEOUS MICROPROCESSOR BEHAVIOUR

5.1. Introduction

The previous chapters of this thesis have proposed and evaluated a model of erroneous behaviour for a selection of microprocessors. This chapter considers methods of exploiting characteristics of erroneous execution as part of a detection strategy. In particular the characteristics of Initial Erroneous Jump (IEJ) and Subsequent Erroneous Jump (SEJ) are identified for this purpose.

The chapter commences by defining the functional allocation of a microprocessor's address space: the used area consisting of program, data, and reserved input/output areas; the unused area consisting of physically implemented and non-existent memory. Detection techniques are then considered for these functional address space allocations. Particular proposals are made using software techniques for the program and physically implemented unused areas of the address space. In instances where a microprocessor does not have its address space totally implemented with physical memory, a proposed hardware unit called an Access Guardian can be implemented to provide detection of unused area access.

The application of fault tolerance incurs a physical and/or performance overhead to the target microprocessor system. Each of the detection techniques considered within this chapter has its required overhead evaluated.

5.2. Address Space Allocation

Within the system memory, areas can be defined which have different execution characteristics dependent on the defined utilization of that memory area. For the purpose of statistical analysis, the memory is divided into functional areas. Each functional area will exhibit a particular instruction distribution.

Initially the address space is divided into two distinct areas, *used area* and *unused area*, such that

$$\left\{ \begin{array}{c} \textit{Total Address Space} \\ \textit{(bytes)} \end{array} \right\} = \left\{ \begin{array}{c} \textit{Used Area} \\ \textit{(bytes)} \end{array} \right\} + \left\{ \begin{array}{c} \textit{Unused Area} \\ \textit{(bytes)} \end{array} \right\} \qquad (5.1.)$$

The unused area is considered to include those address space locations not reserved for, or required by the processor during correct operation. This area can be subdivided depending on the implementation of the resident address space,

- *physically implemented memory, and*
- *non-existent memory.*

The used area contains locations reserved for external communication, and locations for instructions and the data they require for correct operation. The used area can be functionally subdivided into,

- *program area,*
- *data area,* and
- *input/output reserved area.*

The program area is considered to contain all software instructions, opcodes and operands. The data area is considered to contain any information required by the software, i.e. data structures including stacks and linked lists. The input/output reserved area contains those locations specified as reserved for communication ports and exception targets.

The functional allocation of the address space into its constituent areas is shown in Figure 5.1. It should be realized that for microprocessor systems, allocation of the address space rarely involves contiguous functional areas.

5.3. Erroneous Execution in the Unused Area of the Address Space

This section initially describes the character of modelled erroneous execution in the unused area of the address space. In particular the Initial Erroneous Jump (IEJ), first described in Chapter 3, is identified with erroneous execution in the unused area. Software and hardware detection techniques are presented which exploit this characteristic. Finally, the design of the hardware technique is detailed.

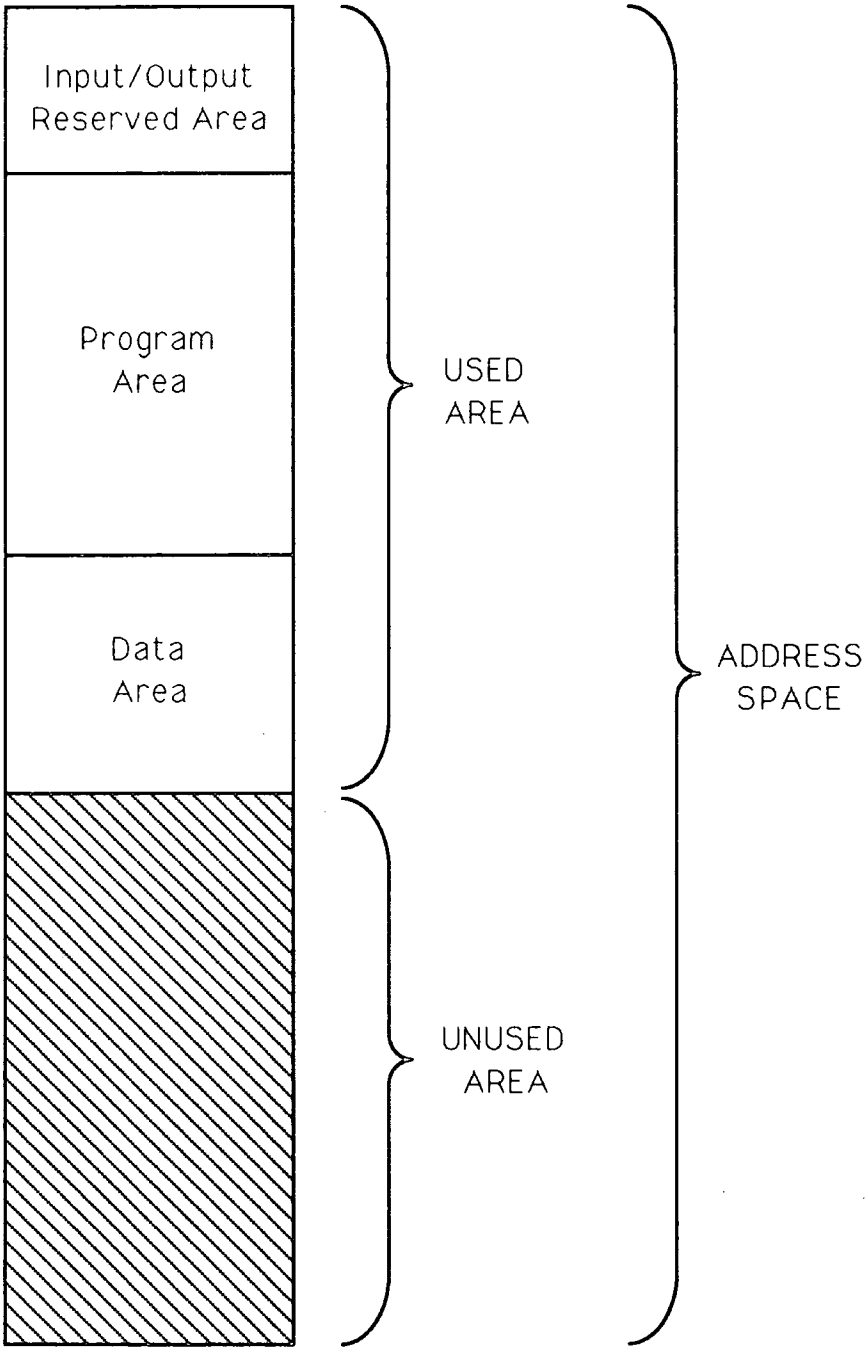


Figure 5.1. : Functional Address Space Allocation

5.3.1. The Initial Erroneous Jump Characteristic

The event associated with the initiation of erroneous behaviour is the Initial Erroneous Jump (IEJ). The destination of an IEJ has been assumed to be a random location in the address space in the model of erroneous execution presented in Chapter 3. Hence the probability that the destination of an IEJ will be in the unused area, $P_{IEJ}(Unused\ Area)$, is dependent on the proportion of the total address space occupied by the used area. That is,

$$P_{IEJ}(Unused\ Area) = \left\{ \frac{Unused\ Area\ (bytes)}{Total\ Address\ Space\ (bytes)} \right\} \tag{5.2.}$$

and,

$$P_{IEJ}(Unused\ Area) = 1 - P_{IEJ}(Used\ Area) \tag{5.3.}$$

A linear relationship between $P_{IEJ}(Used\ Area)$ and *Used Area* for a selection of address space sizes is shown in Figure 5.2. In particular the graph describes the IEJ characteristic exhibited by the Motorola 6800, Intel 8048, and Intel 8085 microprocessors which have a 64 KByte address space, the Intel 8086 microprocessor which has a 1 MByte address space, and finally the Motorola 68000 and 68010 microprocessors which have a 16 MByte address space.

Consider a particular software application on two microprocessors whose only difference is the size of their respective address space. It is evident from equation (5.1.) that the microprocessor with the larger address space will have an unused area occupying a greater proportion of the address space. The IEJ characteristic of equation (5.2.) highlights the profitability of detecting processor execution in the unused area which by definition is erroneous, particular benefit being offered by those microprocessors with a larger address space.

5.3.2. Detecting Erroneous Execution

Access to the unused area of the address space is indicative of erroneous execution. Therefore a technique to detect such access is required to prevent erroneous execution. The unused area may partially or entirely include physical memory locations. These

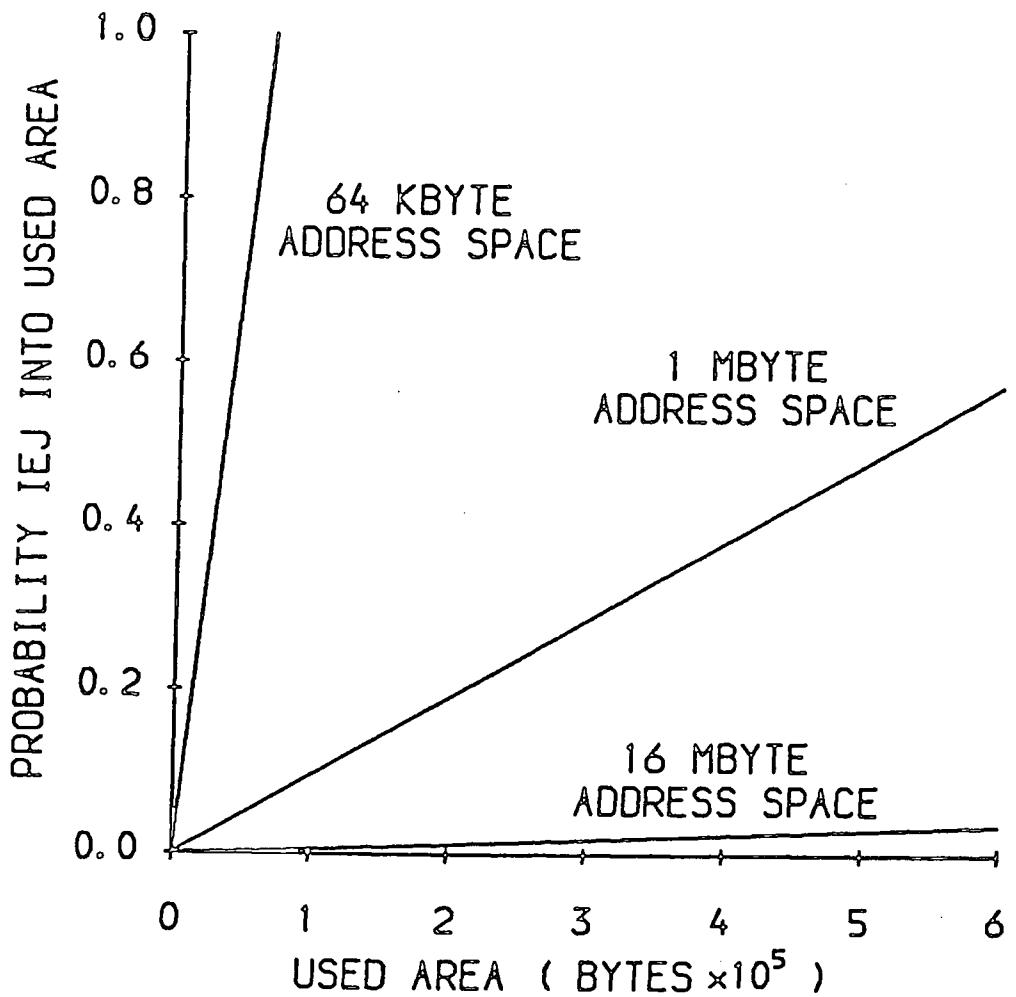


Figure 5.2. : The IEJ Characteristics

locations may occur as a single contiguous block or as a collection of blocks distributed throughout the address space.

5.3.2.1. A Software-Based Technique

Software constructs can be placed in the blocks of physically implemented unused address space to provide detection of erroneous execution. The principles of their structure are as follows:

i) All instructions within the construct should be without an operand requirement in order that there is only one possible program flow path through the memory.

ii) The program flow path of the software construct must have one or more termination points where recovery action is initiated, otherwise no recovery is possible.

The software construct adopted by particular processor systems is influenced by the availability of instructions in their respective instruction sets.

Error latency can be minimized by the placing of restart instructions without an operand requirement at every location in a block of unused physical memory. Processing any one of these restart instructions initiates execution of a recovery routine for the application software. However, not all microprocessors possess such a restart instruction, e.g. the Intel 8048 processor. For microprocessors like these, a software construct called the 'snake' can be used [Pearson, 1983]. The snake construct involves placing a chain of 'no-operation' instructions, without an operand requirement, at each location except the last in a block of unused memory. The terminating location in the block holds a jump instruction. The action of the snake is to 'slide' erroneous execution initiated on it to the jump, which then directs execution to the recovery routine. The error latency associated with the processing time required to 'slide' down the snake can be reduced by placing intermediate jumps, whose destination is the recovery routine, within the blocks. However, in order to preserve detection of erroneous execution at every location in a block of unused physical memory, such intermediate jumps should not require any operands.

Many microprocessor systems do not implement their entire unused area in physical memory, and hence a complementary or alternative detection technique is required for non-existent memory in the address space.

5.3.2.2. Watchdog Timers and Smart Watchdogs

As described in Chapter 2, watchdog timers and smart watchdogs can be implemented in a microprocessor system to detect access to the unused areas of the processors address space.

IBM [1986] describe an analogue watchdog timer which they have developed for microprocessor controllers. It was developed because automatic reset is required when an embedded microprocessor in a controller improperly executes code. Watchdog timers require the application software to correctly reset the watchdog timer. Hence the programmer requires a prerequisite knowledge of the target processor system in order to satisfy the watchdog timer requirements. Such programming practice is much slower than code generation for microprocessor systems employing a transparent fault tolerant technique.

Namjoo & McCluskey [1982] propose a smart watchdog, based on an additional processor, to detect (in a transparent fashion to the application software) unused area access. The watchdog monitors access to the unused area. On an invalid access, the watchdog signals a hardware interrupt to the host microprocessor indicating detection of erroneous behaviour. The host machine then processes the interrupt in a manner that will initiate recovery. It should be realized that the additional processor implemented by the smart watchdog is susceptible to error generation in the way as the microprocessor it is protecting.

5.3.2.3. The Access Guardian Proposal

The redundancy of the smart watchdog can be reduced by implementing its detection function in a dedicated hardware unit. The design of such a unit, referred to as an 'Access Guardian', is proposed here. The Access Guardian provides independent on-line monitoring and detection of unused area access by the microprocessor.

The topology of a microprocessor system incorporating an Access Guardian is shown in Figure 5.3.

The Access Guardian detects whether or not invalid address lines are activated, and if so, impresses an interrupt signal to the microprocessor. This induces a restart state outcome within the microprocessor which then directs execution to a recovery routine for the software application. Implementation of an Access Guardian requires a prerequisite knowledge of the residence of the used area within the microprocessor address space. This implies that Access Guardians can only be used in dedicated microprocessor systems.

The general function of the Access Guardian is shown in Figure 5.4. The Access Guardian takes the system address bus as input to its 'address decoder' which generates a signal when invalid address lines are activated. This signal is then processed with Access Guardian status information by the 'restart generator' to produce an interrupt signal 'RESTART' for the application processor. The interrupt signal must exist slightly in excess of the microprocessor interrupt latency. The interrupt latency is the length of time an interrupt must exist to guarantee processing by the microprocessor. Assuming the interrupt is given highest priority the processor will detect it following the execution of the present instruction. The interrupt signal must therefore be just longer than the longest execution time required by any instruction. A 'timer unit' holds a set interrupt signal for the required period. The detailed design of an Access Guardian is presented in Appendix B.

The effectiveness of detecting unused area access has been investigated using fault insertion experiments. Gunneflo et al [1989] report 60% of faults inserted into an operating Motorola 6809 microprocessor system as generating access to an unused area occupying 88% of the address space. They expect unused area area to fall and rise with decreasing and increasing proportions of the address space occupied by the unused area respectively. This suggestion is supported by Schmid et al [1982] who report only 10% of faults inserted into a Zilog Z80 microprocessor system as generating access to an unused area occupying 10% of the address space.

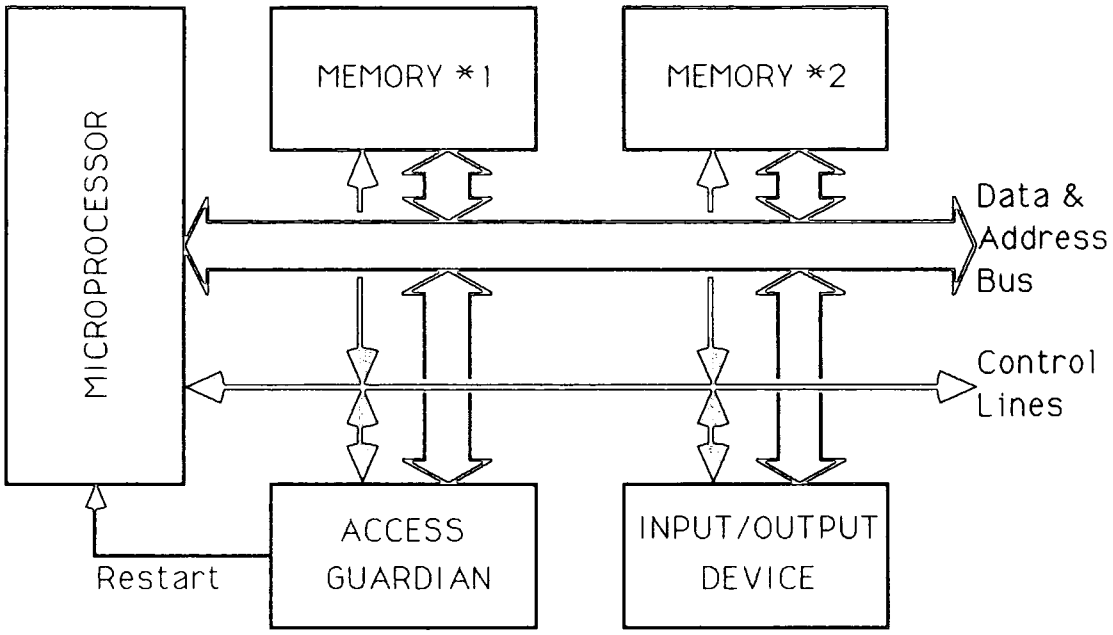


Figure 5.3.: Embedded Access Guardian

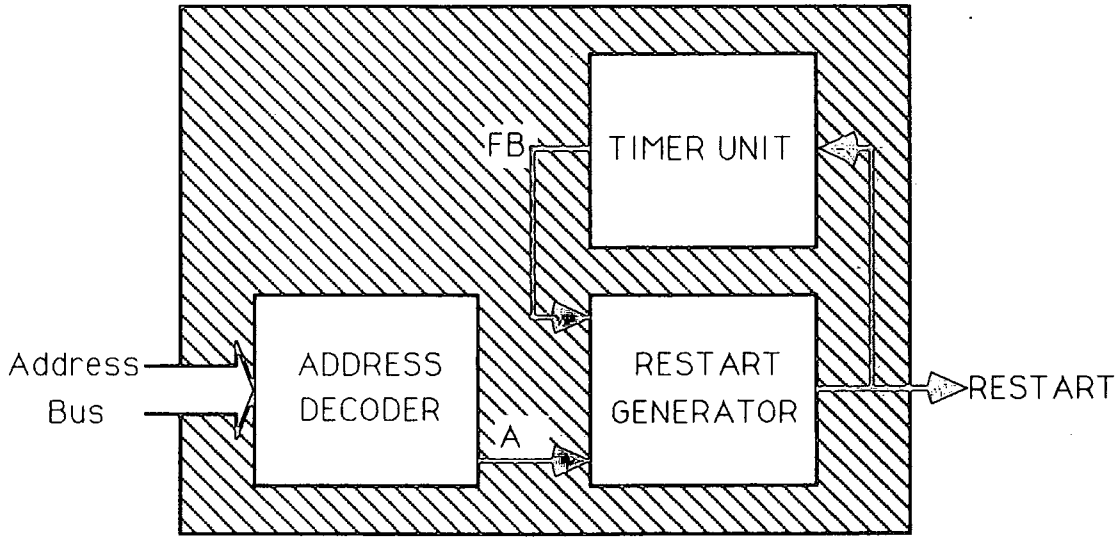


Figure 5.4.: Access Guardian

5.4. Erroneous Execution in Used Area of the Address Space

A technique has been proposed so that erroneous execution in the unused area will be detected. There still remains, however, the possibility of erroneous execution in the used area. This involves periods of linear erroneous execution interspersed by Subsequent Erroneous Jumps (SEJs) within the used area. This characteristic is investigated by tracing successive SEJ destinations, and several microprocessors are evaluated using the model of erroneous microprocessor behaviour proposed in Chapter 3. Techniques are proposed for detecting erroneous execution using software implemented fault tolerance.

5.4.1. The Subsequent Erroneous Jump Characteristic

The probability of a SEJ whose generator and destination are both in the used area can be determined as follows. Let the set $\{L\}$ contain every location in the used area, and N_L be the number of items in the set $\{L\}$. Let the set $\{J\}$ contain every jump type instructions in the microprocessor instruction set, and N_J be the number of items in the set $\{J\}$. Let l be a location in the used area, and j be a particular jump type instruction in the instruction set. Hence $l \in \{L\}$, and $j \in \{J\}$.

Let the function $H(l,j)$ represent the percentage of possible destinations generated by a particular jump type instruction (j) at a used area location (l), that reside within the used area. This function is referred to as the 'hit' function. Let the set $\{T\}$ contain all the locations that can be addressed by a jump type instruction (j) residing at location (l). Then

$$H(l,j) = \frac{Pr(\{T\} \cap \{L\})}{Pr(\{L\})}, \quad \text{where } Pr(\{L\}) = 1, \tag{5.4.}$$

which can be expressed as a conditional probability,

$$H(l,j) = Pr(\{T\}|\{L\}), \tag{5.5.}$$

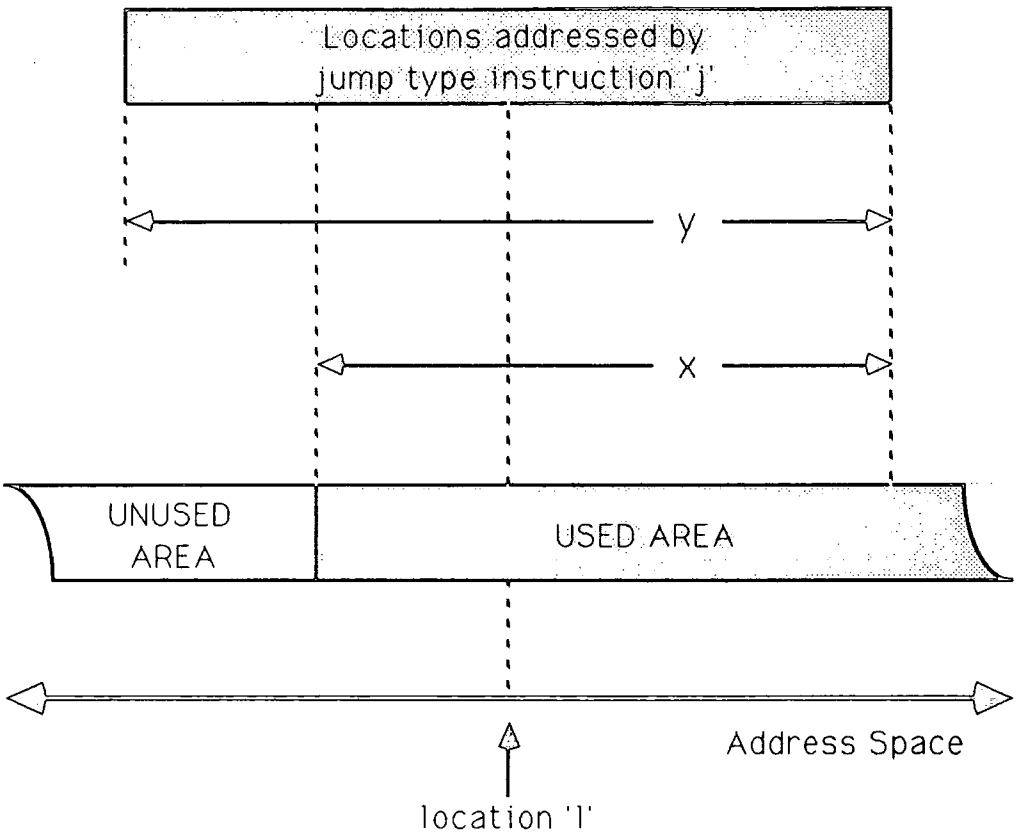
having the boundary condition $Pr(\{L\}) > 0$ is satisfied.

Typically jump type instructions can generate destinations within either a local 2^8, 2^{16}, or 2^{32} byte range. Consider a jump type instruction employing relative addressing with a byte operand specifying the displacement. Various operand values alter the destination of the jump type instruction. If this jump type instruction resides in the middle of the used area of $> 2^8$ bytes then it is guaranteed to generate all its possible destinations within the used area: that is $H(l, j) = 1$. However the same jump type instruction residing in the middle of a used area of 2^7 bytes only has half its possible destinations within the used area: that is $H(l, j) = 0.5$. Figure 5.5. is provided to further aid comprehension of the HIT function.

Let P_{SEJ}(Used Area) be the probability of a jump type instruction (j) selected at random from the microprocessor instruction set, residing at a random location in the used area (l), generating a destination which is also within the used area.

$$P_{SEJ}(Used\ Area) = \frac{1}{N_J.N_L} \cdot \sum_{j \in \{J\}} \sum_{l \in \{L\}} H(l, j). \qquad (5.6.)$$

The relationship between $P_{SEJ}(Used\ Area)$ and *Used Area* for a range of 8, 16 and 32-bit microprocessors is investigated in Figure 5.6. All jump type instructions within the 8-bit Intel 8085 microprocessor instruction set specify absolute target addresses and hence its SEJ characteristic is linear. When half the Intel 8085 processors address space is used, there is a 50% expectation that a SEJ in the used area will return to the used area. For the same address space utilization, however, the 8-bit Motorola 6800 microprocessor has an 80% probability that a SEJs generator and destination lie within the used area. This is because the Motorola 6800 processor instruction set does have some relative addressing mode jump instructions. Similar results are obtained for the 16-bit processors evaluated. The Motorola 68000 microprocessor has a much higher probability of a used area SEJ targeting the the used area again than the Intel 8086 because it has a far higher proportion of opcode formats within its instruction set dedicated to relative addressing. For example, a 300 KByte used area on the Motorola 68000 and Intel 8086 have respective probabilities of approximately 95% and 70% that a SEJ generated in the used area will also target the used area. The 32-bit microprocessors evaluated show that for a 300 KByte used



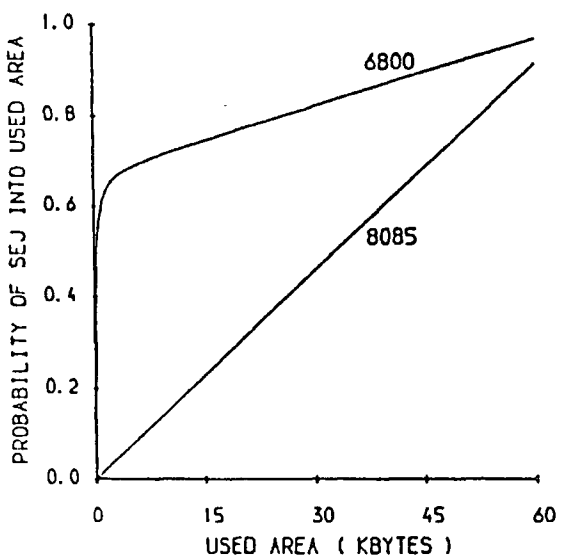
Let 'y' be the number of location addressed by jump type instruction 'j'.

Let 'x' be the number of locations 'y' that fall within the used area when then jump type instruction 'j' resides at location 'l' in the used area.

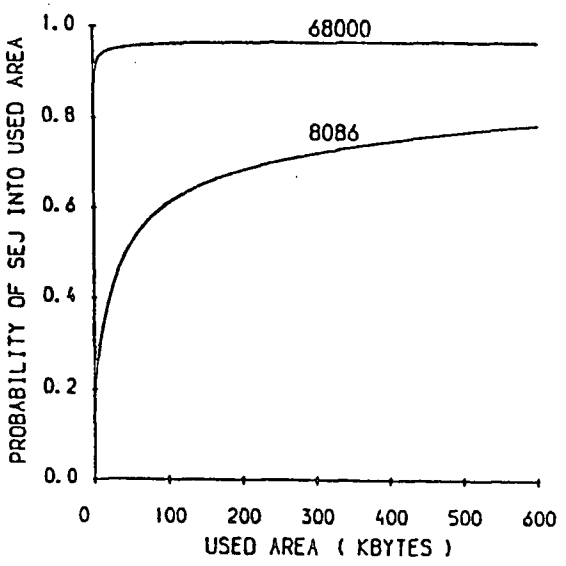
The HIT function of the jump type instruction at location 'l' in the used area is then defined as

$$H(l,j) = \frac{x}{y}$$

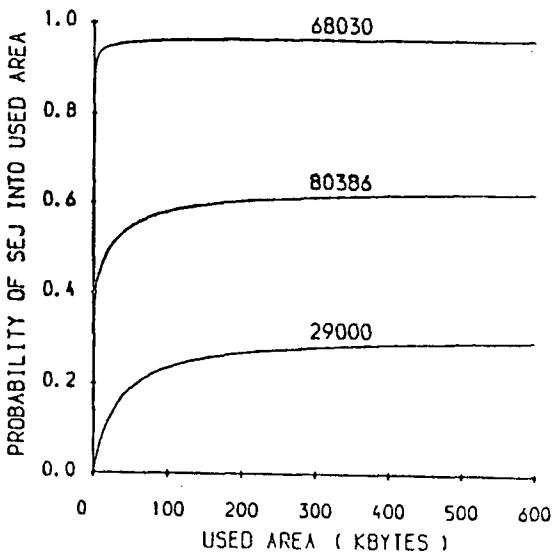
Figure 5.5. : The HIT Function



(a) 8-Bit Microprocessors



(b) 16-Bit Microprocessors



(c) 32-Bit Microprocessors

Figure 5.6. : The SEJ Characteristic

area of random data, the AMD Am29000 Intel 80386, and Motorola 68020 microprocessors are expected to generate a target with the used area of approximately 25%, 60%, and 95% respectively. The variation in the HIT function evaluation for the processors is due to the proportion of relative jump opcode formats in their instruction sets.

A high probability of an SEJ in the used area generating a target address which is also in the used area suggest extended periods of erroneous execution in the used area. Such behaviour can be extremely hazardous, not only involving mal-operation but also program area corruption from invalid data manipulation. Therefore, a method of detection is required within the used area to prevent extended erroneous execution.

5.4.2. Detection Using Software Implemented Fault Tolerance

The following section describes techniques which can be implemented in the software to detect erroneous execution in the used area. The techniques utilize instructions that generate a restart outcome. Such instructions direct execution to a predefined location in memory at which a recovery routine resides. The recovery routine restores operation as required by the application software, two possible recovery strategies are reset and rollback.

5.4.2.1. Program Areas

A software detection technique is proposed which exploits the SEJ characteristic. In particular, erroneous jumps are considered which are generated by invalid interpretation of an operand as an opcode. Such erroneous jumps are referred to as invalid branches. Detection mechanisms are strategically inserted at each identifiable invalid branch destination.

Detection Mechanism Construction

The actual construction of a detection mechanism will vary in detail for different microprocessors. A detection mechanism consists of an initial relative branch instruction over the remainder of the detection mechanism so that logical control flow of

the correctly executing program is not interrupted. The remainder of the detection mechanism consists of a number of 'seed' instructions.

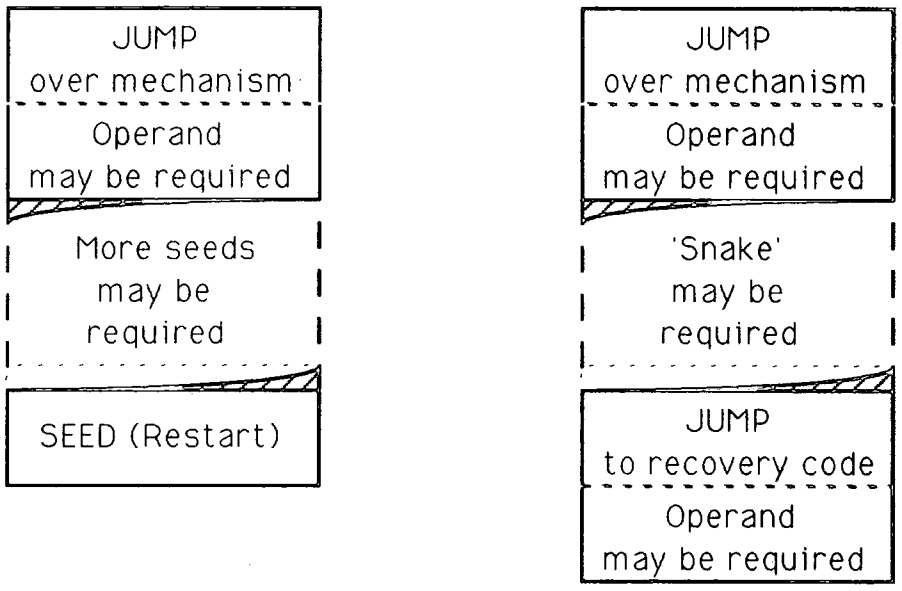
The instructions used to construct a detection mechanism should not require any operands, however, this may not always be possible. In such instances care must be taken to avoid the use of operand bit formats which when erroneously interpreted as an opcode generate erroneous jumps. Failure to ensure that detection mechanisms do not themselves generate erroneous jumps leads to successive detection mechanism placements with no guarantee of placement completion.

A detection mechanism seed instruction is a software exception without operands which, through a restart outcome, directs execution flow to a recovery routine. The number of seeds required depends on the individual placement of a detection mechanism.

General examples of detection mechanism structure are shown in Figure 5.7. The Motorola 68000 microprocessor facilitates detection mechanism constructs like that in Figure 5.7(a), the jump over mechanism being provided by the operandless hexadecimal instruction 600X where 'X' is the necessary relative displacement, and the seed is provided by the hexadecimal instruction 6001 which is branch that always generates an 'odd address' exception (restart) and again does not require any operands. Within the Intel 8048 microprocessor there are no restart instructions or jump instructions without a operand requirement. The detection mechanism in this case is like that in Figure 5.7(b). The mechanism uses the hexadecimal instruction 04XX for the jump over the mechanism, the hexadecimal no-operation instruction 00 for the snake, and the hexadecimal instruction 04XX to jump to the recovery routine (mimic restart function) where 'XX' specifies the hexadecimal representation of the jump destination location.

Detection Mechanism Placement

Figure 5.8. shows the two basic methods of inserting a detection mechanism within the target software. Forward invalid branch destinations are covered by inserting the detection mechanism at the location of the preceding instruction.



a) Instruction Set includes restart instructions that do not require operands. The instruction set includes jump instructions.

b) Instruction Set does not have any restart instructions, but has 'no-operation' instructions that do not require operands. The instruction set includes jump instructions.

Figure 5.7. : Detection Mechanism Constructions

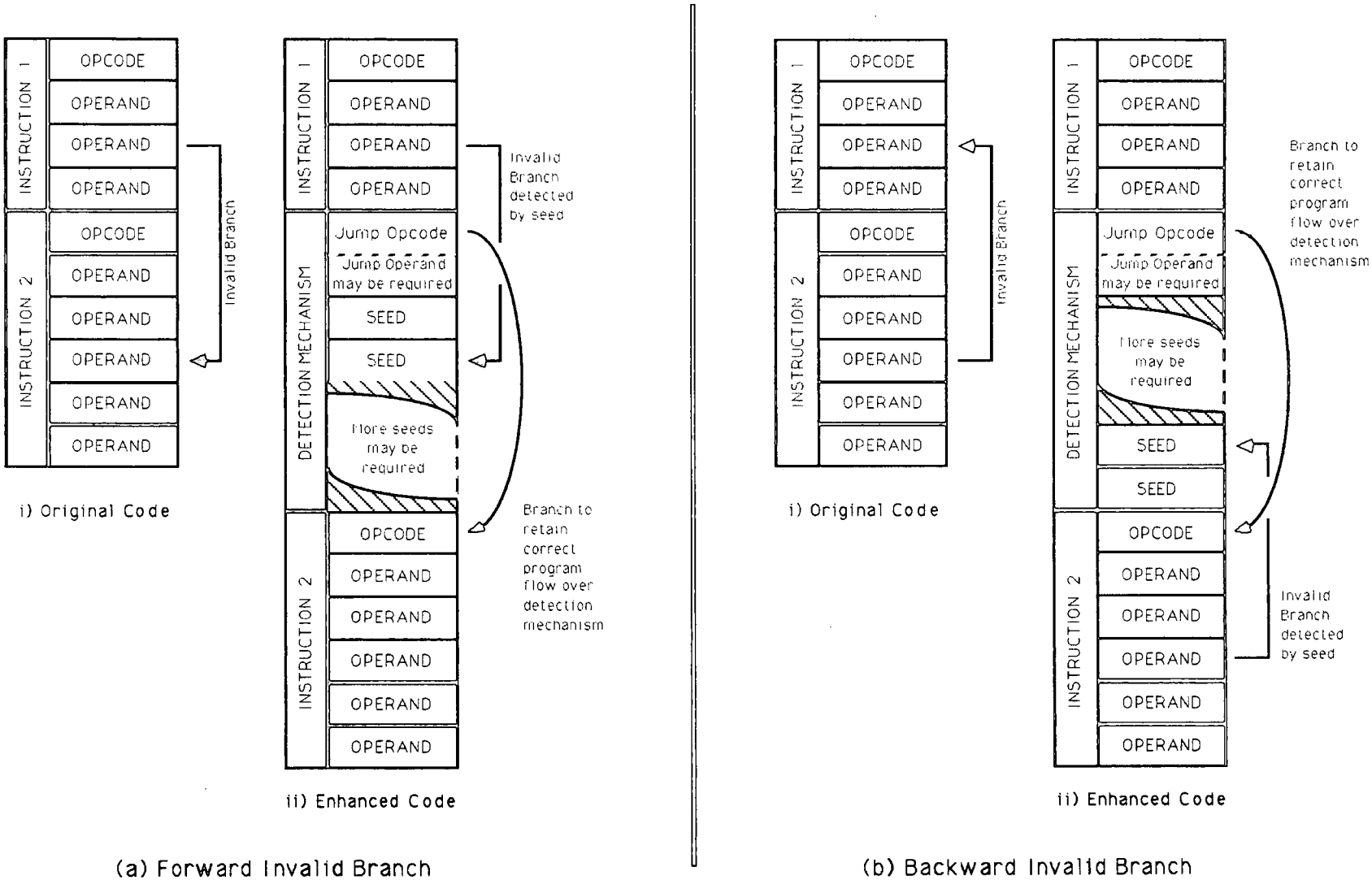


Figure 5.8. : Detection Mechanism Placement

Detection mechanisms are inserted to cover backward invalid branches at the location of their respective following instruction. Each detection mechanism requires sufficient seeds to ensure that the invalid branch destination has a resident seed. The maximum number of seeds required will be equivalent to the byte length of the largest instruction construct in the instruction set.

The insertion of a detection mechanism may itself alter the destination of an invalid branch. This situation arises when the invalid branch generator has a specified displacement contained in the byte locations directly following the host instruction and a detection mechanism is inserted immediately after the host instruction. A good example of this is provided by the Motorola 68(7)05 microprocessor where the 'test and branch' instruction requires two operands, the final operands containing the relative displacement. This instruction has the same size as the maximum length instruction within the instruction set; therefore, whenever an invalid branch has this operation, its destination information will always be contained outside the generating instruction. If a detection mechanism is inserted immediately after the generating instruction then the destination for the invalid branch is altered by the change in the information content at the location specifying the relative displacement. Successive invalid branches can be generated in this manner. Application of detection mechanism placement should test for this situation and take evasive action to prevent changing the destination of an invalid branch.

The Detection Capability of the Mechanism

The mechanisms inserted within the program code provide a detection capability which has two methods of activation. The first method of detection is associated with an invalid branch generated by erroneous execution. Detection mechanisms are placed in order to detect such SEJs. Secondly, ensuing linear erroneous execution processing through a placed mechanism is detected when a seed is interpreted as an opcode. This second method of detection is guaranteed to be successful if the detection mechanism has a number of seeds equivalent to the byte length of the longest instruction within the application processors instruction set.

Detection Mechanism Placement Deadlock

Detection mechanisms cannot be placed where the generator and destination of an invalid branch are both operands of a single instruction. This occurrence is referred to as *placement deadlock*. There are two particular types of placement deadlock, those where the invalid branch has a forward direction and those with a backward direction. In addition placement deadlock occurs when the destination of an invalid branch is displaced forward of its generating instruction by the equivalent or fewer bytes than that required by the detection mechanism's jump instruction. Placement deadlock with a backward invalid branch is critical because an infinite execution loop may be created. This processing outcome is classified as a failure and has a particular hazard in real-time systems. Placement deadlock involving a forward branch does not share this hazard, erroneous execution continuing unhindered through the associated code.

5.4.2.2. Data and Reserved Input/Output Areas

Data areas and areas reserved for input/output communication can be manipulated using similar techniques to facilitate a detection capability of erroneous execution. The information content of both area types cannot be changed, but the method and location of storage can be altered.

Halse [1984] investigates methods of inserting special sections of code within the data area. Erroneous execution in the data area is detected when it flows through one of these sections of inserted code. The efficiency of various sizes of code insertions with different desperations throughout the data area are analysed by Halse. This technique is not transferable to the reserved input/output area because the locations of this area are fixed and no code insertion is therefore possible.

An alternative technique involves utilizing particular bits of each memory element to specify an opcode restart operation. The remainder of the memory element is free for information storage. This technique can best be explained using the Motorola 68000 as an example. Within this processor the hexadecimal opcode format FXXX can be used where 'X' denotes an unspecified content. This opcode format is not necessarily available in upwardly compatible members of the Motorola 68000

family, e.g. the Motorola 68020 reserves this format for a co-processor. The execution of a Motorola 68000 FXXX opcode is defined to be illegal and to generate an exception (restart outcome), and hence can be used to detect erroneous execution. The unspecified opcode bits can be used to contain useful data area or input/output reserved area information. Implementation of this technique in the data area and input/output reserved areas will required the application software only to extract the least significant 12-bits of information from each memory location. In addition, the input/output locations should have the most significant 4-bits hard-wired for the restart opcode format. This technique incurs redundancy, which may be substantial in some microprocessor applications. Indeed, the technique may not be feasible for some microprocessor systems.

A third technique involves moving the location of data areas in the address space so that their address specification within an operand has the format of a restart instruction. This ensures that when erroneous execution in the program area generates a SEJ destination in the data area, erroneous execution is detected. Application of this technique is highly dependent on the host microprocessor instruction set. The best results are obtained for those instruction sets with a larger number of restart generating opcode formats. Small numbers of restart opcode formats restricts the number of locations available for positioning sections of data area. This technique is investigated further by Halse [1984]. Again, however, this technique cannot be applied to the input/ output reserved area because its locations are fixed, although some locations may be available that, by coincidence, map restart opcode formats.

5.5. The Overheads of Implementing Fault Tolerance

5.5.1. Hardware Fault Tolerant Techniques

The Access Guardian and smart watchdog have an error latency much smaller than that typically exhibited by watchdog timers. In addition their application, unlike the watchdog timer, is transparent to the application program. The use of an Access Guardian or smart watchdog, therefore, releases the application programmer

from requiring a knowledge of the target processor system architecture in order to produce dedicated software.

The design of the Access Guardian can be applied to most microprocessor systems. The complexity of the 'address decoder' is dependent on the nature of the address space, and the bus architecture employed by the microprocessor. The 'timer unit' will also vary in size depending on the microprocessor interrupt latency. The 'restart generator' has a fixed size. An Access Guardian designed in Appendix B requires 60 logic gates, representing 17 standard TTL IC parts. This represents a significant reduction in the gate overhead introduced by a smart watchdog which typically has thousands of gates.

The Access Guardian acts in parallel with the microprocessor and does not inflict a performance overhead to the system during correct operation. Its reduced size in comparison with a smart watchdog also implies a smaller chance of the 'doomsday' syndrome occurring by which the hardware detection unit fails [Damm, 1988].

5.5.2. Software Implemented Fault Tolerant Techniques

The overheads of used area software enhancement are additional memory requirement and increased operational processing during correct operation of detection mechanism jumps. Modification of the unused physical memory locations does not incur any overhead to the microprocessor system because during the course of correct operation the section of the address space is totally independent of processor action.

The memory extension required by the software implemented fault tolerant technique proposed for the program area can be reduced by generating optimum size detection mechanisms at each placement rather than a default maximum size. Whilst this reduces the memory overhead, it also decreases the effectiveness of the mechanism to detect linear erroneous execution. The relative cost of a byte of physical memory has decreased over recent decades [Freer, 1987] and, therefore, the memory overhead is not predicted to be a major system constraint. Nevertheless, in those systems with a limited memory, optimum size detection mechanisms can be implemented.

A memory overhead is also produced by the insertion of software in the data area to detect erroneous execution. Further details of the expected overhead for particular

processor systems can be found in Halse [1984]. This memory overhead does not have an associated performance overhead.

The extra processing requirement of detection mechanism jumps during correct operation of program code may prove critical for some stringent real-time and parallel processing applications. This processing overhead is not influenced by the placement of optimum or default size detection mechanisms in the target code. However, for the majority of applications this overhead is considered to be acceptable.

The technique proposed for the data and reserved input/ output area by which the data content of each memory location is reduced in order to give that location a detection capability, also generates a processing overhead. Data transfers and memory requirement may be increased. The magnitude of this overhead is application dependent. The architecture of a microprocessor could incorporate this technique in order to reduce lost operational performance.

5.6. A Fault Tolerant Strategy for Microprocessor Controllers

The detection techniques described in this chapter cover attributes of erroneous execution associated with the model of erroneous microprocessor behaviour presented in Chapter 3. Individual application of one of the techniques will improve the reliability of the processor system in relation to this mode of failure. However, the reliability of the system can be further improved by the selection of techniques for collective implementation. Such techniques should be complementary and feasible for incorporation into a particular microprocessor system. Hence, the selection of fault tolerant techniques is termed 'strategic'.

The model of erroneous microprocessor behaviour described in Chapter 3 is summarized in Figure 3.2. This figure can now be modified to include the detection capability of unused area access via the Access Guardian, and invalid used area operation via the detection mechanisms planted in the software. The model of erroneous behaviour in a microprocessor implementing such fault tolerant techniques is shown in Figure 5.9.

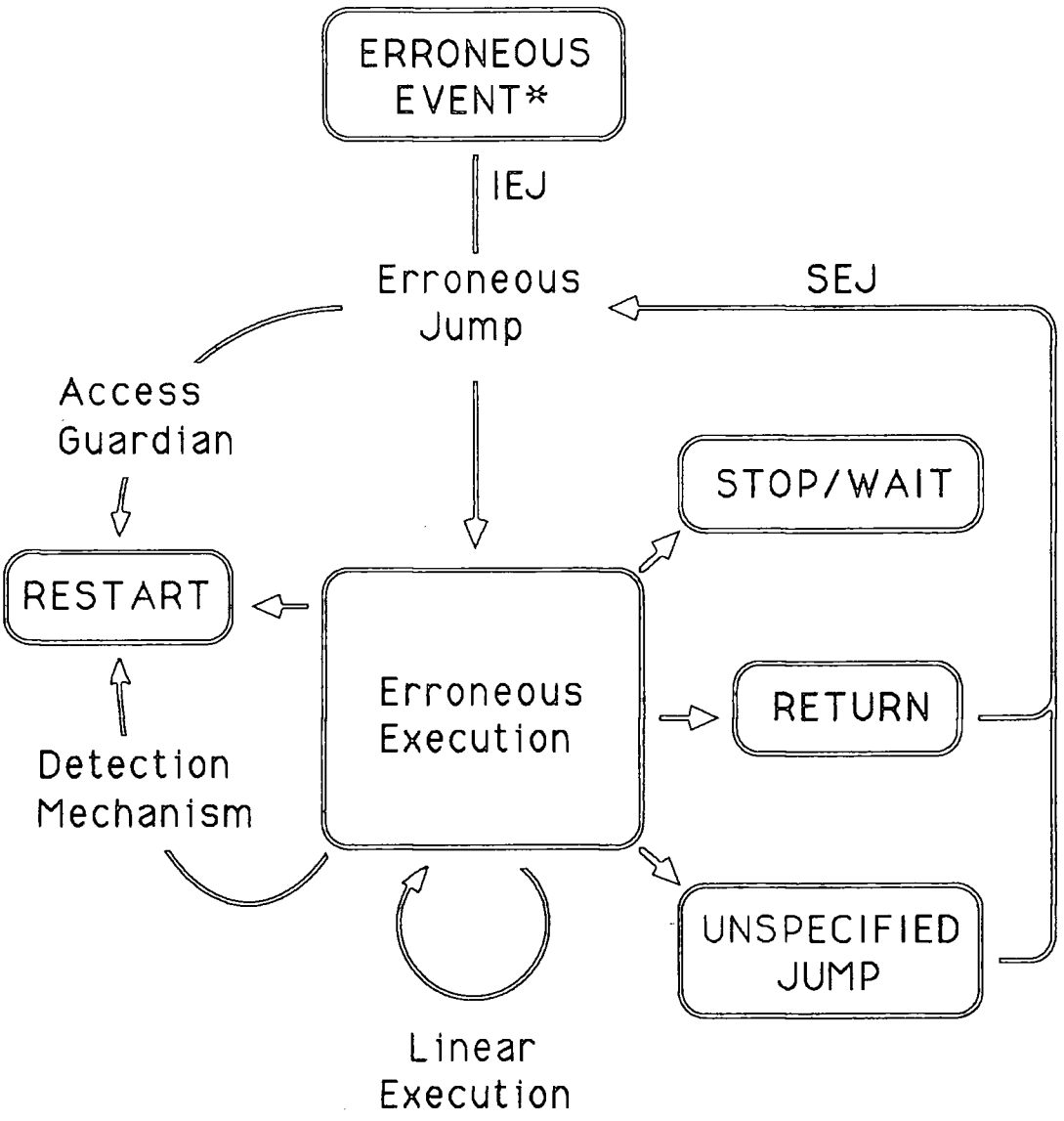


Figure 5.9. : Erroneous Execution Model :
Enhanced Transient Fault Recovery
(* manifested transient fault)

The reliability of a selection of microprocessors implementing an Access Guardian are shown in Figure 5.10. The evaluations assume a worst case of no recovery capability exhibited by the used area so equation (3.49.) with equation (3.26.) substituted becomes,

$$R(t) = exp\{-q.P(E_f).t\}, \tag{5.7.}$$

where,

$$P(E_f) = \left\{ \frac{\text{Used Area (bytes)}}{\text{Address Space (bytes)}} \right\}, \tag{5.8.}$$

and q is the event rate, $P(E_f)$ the probability that the event initiates erroneous execution, and t is time.

Figure 5.10. has a normalized time base. Reliability can also be expressed as MTTF using the following equation for the microprocessor system described above,

$$MTTF = \frac{1}{q}.\left\{ \frac{\text{Address Space (bytes)}}{\text{Used Area (bytes)}} \right\} \tag{5.9.}$$

Table 5.1. shows the MTTF calculations corresponding to Figure 5.15. where events initiating erroneous microprocessor behaviour are taken to occur once a month (every 714 hours).

The complementary application of fault tolerance in the used area enables the reliability of the microprocessor system to be improved. Future chapters investigate the recovery capability enhancement realized by particular microprocessor systems. Meanwhile it is sufficient to demonstrate the benefit of detecting erroneous microprocessor behaviour in the used area of the address space. Consider a Motorola 68000 microprocessor system with 48 KBytes used area and implementing an Access Guardian. The used area is initially assumed not to have a recovery capability. For the purpose of statistical analysis let the used area be divided into program and

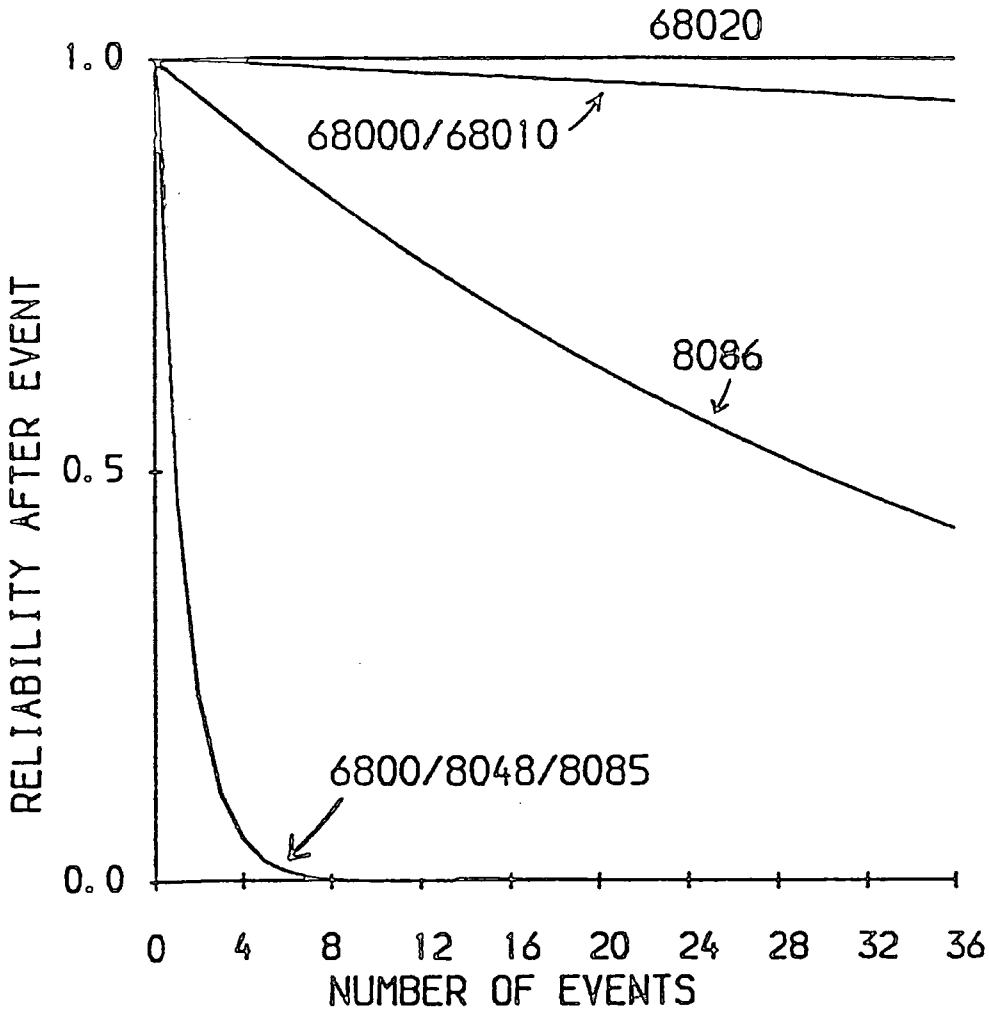


Figure 5.10. : **Microprocessor Reliability with Access Guardian**
(Used Area = 48 KBytes)

Microprocessor	Class	$P(E_f)$	MTTF
6800	8-Bit	0.750000	952 hrs = 39 days
8048	8-Bit	0.750000	952 hrs = 39 days
8085	8-Bit	0.750000	952 hrs = 39 days
8086	16-Bit	0.046870	15232 hrs = 21.6 months
68000	16-Bit	0.001465	487567 hrs = 56 yrs
68010	16-Bit	0.001465	487567 hrs = 56 yrs
68020	32-Bit	0.000006	119047619 hrs = 13590 yrs

Table 5.1. : MTTF with Unused Area Detection

Program/Data	$P(E_f)$	MTTF
48K/0K	0.001465	487567 hrs = 56 yrs
40K/8K	0.001383	516476 hrs = 59 yrs
8K/40K	0.001052	678979 hrs = 78 yrs

Table 5.2. : MTTF Enhancement with Used Area Detection

data areas with no and total recovery capability respectively. Reliability enhancement is now dependent on the proportion of the used area occupied by the data area. Table 5.2. notes 8 KByte and 40 KByte data areas and calculates their respective MTTF influence on the microprocessor system. These calculations are represented as reliability curves in Figure 5.11. Of course the program area can have a recovery capability too: the above calculations are given purely as an example.

The strategic selection of fault tolerant techniques is not limited to those specialized techniques presented in this chapter for particular modes of failure attributed to erroneous microprocessor behaviour described in Chapter 3. Additional techniques such as Recovery Blocks for programs, and parity bit checking for physical memory (both described in Chapter 1), can be incorporated to further enhance reliability through the coverage of other modes of processor system failure.

5.7. Summary

Operational characteristics have been identified in the model of erroneous microprocessor behaviour proposed in Chapter 3. In particular the characteristics of Initial Erroneous Jump (IEJ) and Subsequent Erroneous Jump (SEJ) are associated with erroneous execution within the unused and used areas of the processor address space respectively. The characteristics are modelled for a selection of 8-bit, 16-bit and 32-bit processors, and variations are observed with differences in processor architecture and instruction sets. Detection techniques are proposed which exploit these characteristics in order to provide fault tolerance and hence increased reliability to the microprocessor system.

A detection capability can be provided for the unused area using either a software based technique for physically implemented memory, and/or an Access Guardian which provides an additional detection capability for non-existent memory.

The used area incorporates functional areas for the program, data, and input/output communications, and can be implemented in volatile and non-volatile memory. A detection capability is provided for the program area by the application of a proposed technique involving the strategic insertion of mechanisms at invalid branch

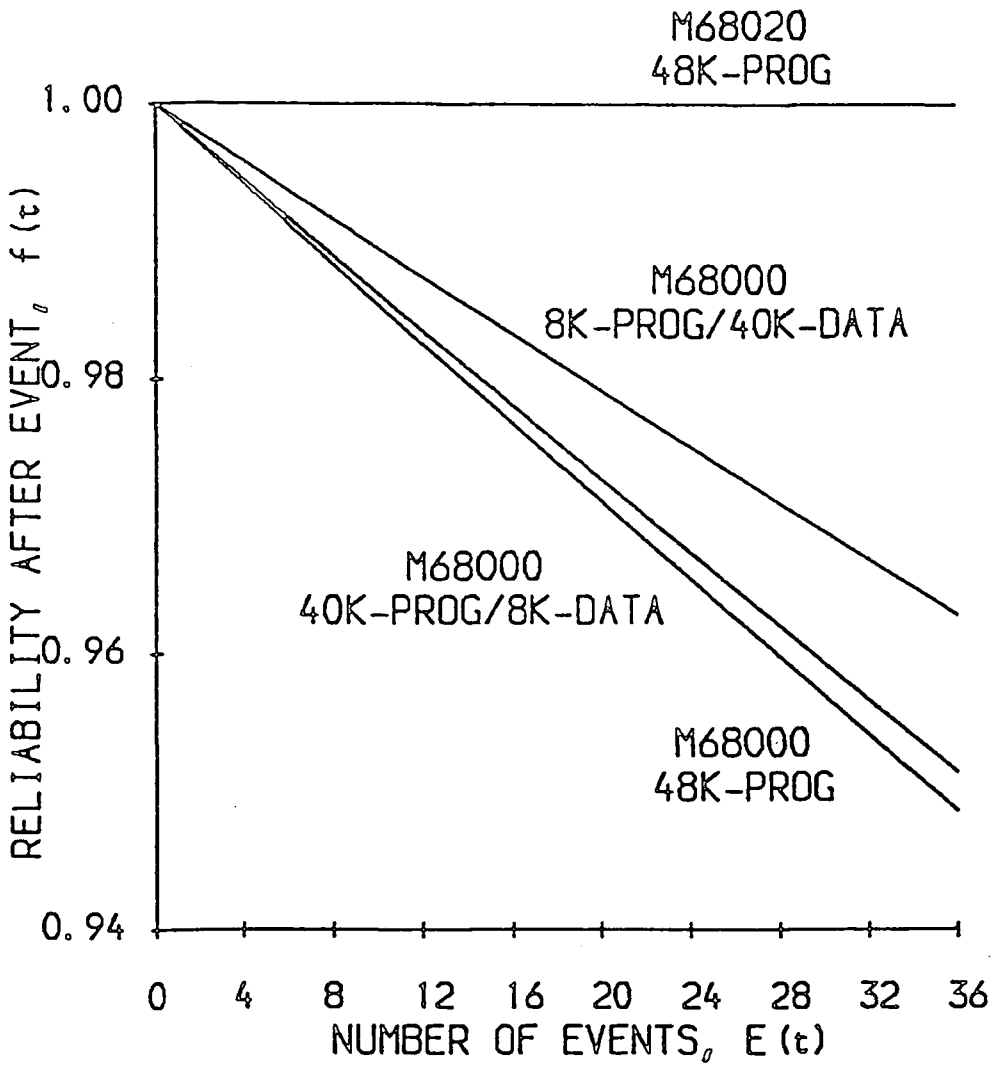


Figure 5.11. : Enhanced Microprocessor Reliability with Access Guardian and Used Area Detection Capability

(SEJ) destinations to detect erroneous execution. The application of this technique and its performance evaluation are covered by future chapters. Other techniques are discussed in relation to providing a detection capability for the data and reserved input/ output areas of the used area.

Application of fault tolerance in the used area involves an additional memory overhead. This memory overhead inflicts a processing performance overhead when inserted within the program area, namely the jumps over detection mechanism which avoid correct program flow corruption. The magnitude of these overheads is evaluated for particular applications in the following chapters.

Fault tolerant techniques can be strategically selected for collective application in order to achieve high reliability. The selection criteria used depends on the fault classes which require detection and the feasibility of implementing fault tolerant techniques within particular microprocessor systems.

Chapter 6

POST-PROGRAMMING, AUTOMATED, RECOVERY UTILITY (PARUT)

CHAPTER SIX

POST-PROGRAMMING, AUTOMATED, RECOVERY UTILITY (PARUT)

6.1. Introduction

A prototype software utility has been designed to automatically apply the software implemented fault tolerant technique, proposed in the previous chapter, on program code. This utility, called PARUT (Post-programming Automated Recovery UTility), can also apply other software based fault tolerant techniques.

This chapter initially outlines the objectives of PARUT, and these are appraised at the end of the chapter to assess the success of the prototype. PARUT, its function and structure, are described in overview, a description of the physical mechanics of the code can be found by examining the annotated utility listing in Appendix C. Finally, enhancements for the PARUT prototype are suggested, and a proposal for the development and application of PARUT as a standard programming tool discussed.

6.2. Design and Development Objectives for the PARUT Prototype

The software implemented fault tolerant technique proposed in Chapter 5 involves inserting detection mechanisms at machine code level to cover invalid branches associated with erroneous microprocessor behaviour. Manual application of the technique can be complex, especially for large target programs. The PARUT prototype is designed to automate the technique's application.

As a prototype, PARUT does not have rigorous specification but rather a set of objectives. In order to facilitate a wider application of PARUT, its initial objective is broadened to include those listed below.

- Apply and assess software implemented fault tolerant techniques.
- Process software for a variety of target microprocessors.
- Facilitate application to any software whose host processor is supported.
- Produce a report assessing utility activity.

In addition to the design objectives it is worthwhile specifying some development objectives for the production of this prototype utility. In particular the prototype program should exhibit qualities facilitating programmer/analyst comprehension and modification of module mechanics. These qualities are especially important in the production of a prototype because alterations are commonplace.

6.3. A Functional Overview of PARUT

PARUT has two input requirements: a copy of the software and a description of the microprocessor on which it resides. The microprocessor description input to PARUT is a file, referred to as MICRO_FILE, containing the target processor instruction set. The file lists the defined instructions within the instruction set, specifying each instruction opcode. The software presented to PARUT for processing is in machine code format because the detection capability assessment and fault tolerant technique application require knowledge of the opcode and operand usage on the target processor. The target software is held in a file called CODE_FILE.

The execution of PARUT generates a report file, referred to as ANALYSIS_FILE, which documents the detection capability assessment of the software under investigation. The detection capability is evaluated by determining the proportion invalid branches that are detected during erroneous execution. PARUT also produces a file called RESULT_FILE, containing the fault tolerant version of CODE_FILE, when a software implemented fault tolerant technique is applied to the target software represented in CODE_FILE. The format of this file can be tailored to meet specific requirements. PARUT currently outputs the enhanced software in a format which facilities easy user interpretation of the utility action at machine code level.

An overview of the PARUT program is shown in Figure 6.1. Examples of the two input files, MICRO_FILE (Motorola 68000 microprocessor) and CODE_FILE, and the two output files, RESULT_FILE and ANALYSIS_FILE, associated with the operation of PARUT on CODE_FILE can be found in Appendix C.

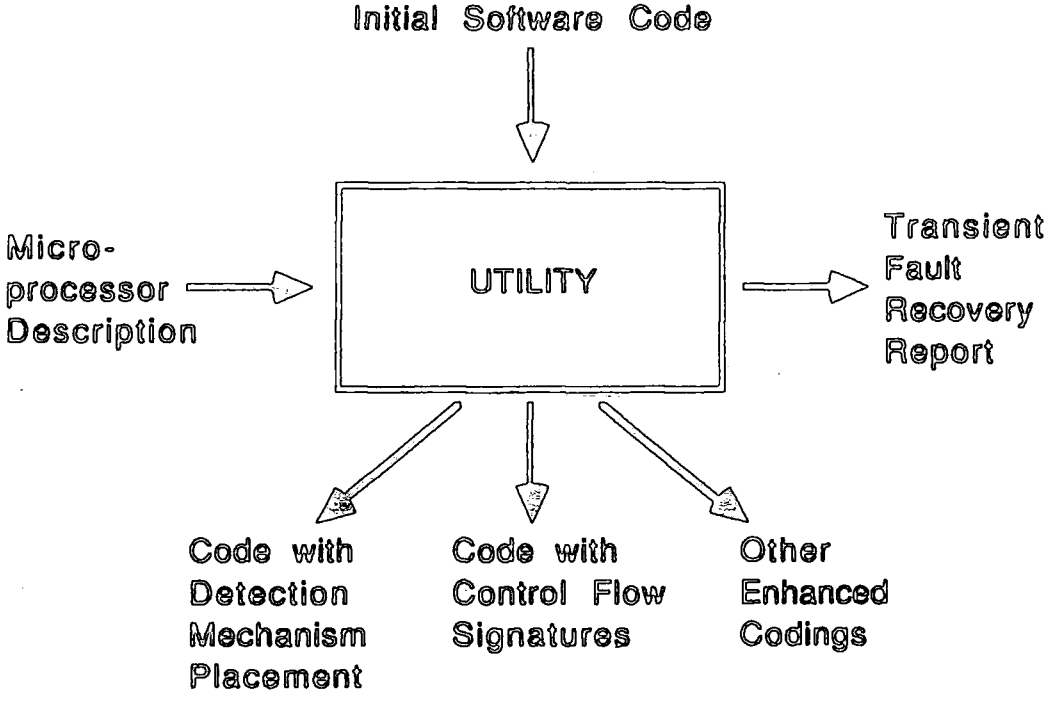


Figure 6.1. : PARUT Overview

6.4. Design Features Incorporated into the PARUT Prototype

Particular design features can be incorporated into the PARUT program to realize prototype objectives, outlined in section section 6.2., or to underpin the procedural activity of PARUT which is described in the next section of this chapter. This section presents such design features and describes the criteria for their application.

6.4.1. Programming Language

PARUT is implemented in the Pascal programming language. This language was selected for the prototype because of its structural constructs and readability. An alternative considered was the programming language C, but was rejected because language constructs such as those involving linked lists are difficult to understand when the reader is not proficient in the language. An important feature of a prototype program is readability. A future development of PARUT might involve the translation of the present code into another language deemed more appropriate. It is considered that Pascal is relatively easy to translate.

6.4.2. Programming Style

The utility program has been written implementing 'good' programming practice [Sommerville, 1985]. This involves developing code in concise modules (a few tens of lines) which exhibit low coupling and high cohesion. Coupling and cohesion refer to the required passing of external parameters to a module, and unity of operation respectively. Such programming practice facilitates easy modification or replacement of modules without disruptive consequences for the remainder of the utility program. Furthermore, 'good' programming practice also encourages the production of readable code.

6.4.3. The Diagnostics Facility

A diagnostics facility is provided to aid understanding of the function of PARUT. The facility is activated by setting the 'DIAGNOSTIC' variable at the beginning of the PARUT listing to 'true'. When active, the facility generates a file called TRACE_FILE in which all functions and procedures accessed by the code operation

insert an entry specifying their name. Nested functions appear in the TRACE_FILE as indented entries. This file can be accessed by the analyst to monitor the chronological activity of PARUT. The last enclosure in Appendix C is a typical TRACE_FILE.

6.4.4. Target Software

PARUT processes target software at machine code level in order to apply and/or analyse its fault tolerance. The data structure chosen to represent the machine code is a linked list of records. This data structure is used because it requires no predefinition of dimensions and can easily be manipulated when inserting records representing code associated with the application of fault tolerance.

Each record within the linked list contains information describing the characteristic of a machine code element (usually 8 or 16 bits). The contents of a record are itemized below and can be found at the beginning of the PARUT listing in Appendix C under the 'TYPE' declaration. The items within each record are reviewed below :

next_address & last_address

- are pointers connecting adjacent records in the linked list. 'next_address' and 'last_address' are set to nil in the last and first records in the linked list respectively denoting the lists termination.

op, optype, & address

- specify the absolute value of the machine code element (typically 8 or 16 bits), whether it is an opcode or operand, and its resident location in the address space.

offset

- is a parameter used in address processing.

seeded

- specifies the status of an operand identified as a potential invalid branch ; 'true' and 'false' signify the presence and absence of fault tolerance respectively.

offset

- is a parameter used in address processing.

jump_type, jump_too & jump_address

- specify the type of jump instruction that the item 'op' generates under erroneous execution as an opcode (details are shown under the 'TYPE' declaration at the beginning of the PARUT listing), a pointer to the target location, and the target address respectively. Non-jump instructions set 'jump_type', 'jump_too', and 'jump_address', to '0', 'nil', and '0' respectively.

jump_from

- is a pointer to a record containing machine code identified as potentially generating an invalid branch whose target is this record, the pointer has the default setting of 'nil'.

6.4.5. Target Processors

Target software is input to the utility via a file called CODE_FILE. PARUT processes the software at source code level. The source code on most microprocessor systems is not directly readable and hence an indirect method of input for the code is required. CODE_FILE contains data generated by UNIX 'adb' (a debugger). Particular details of CODE_FILE can be found in Appendix C. In summary, the file contains two sections: a memory dump of the resident source code, and a list of opcode addresses within the source code.

6.5. A Description of the PARUT Prototype's Operation

This section briefly describes the procedural activity of, and user interaction with, PARUT. An annotated listing of the PARUT code is held in Appendix C.

Within the program structure defined by Pascal, the root module is a function called 'MAIN'. A call chart of functions and procedures used by MAIN is shown in Figure 6.2. The chart depicts module operations in rectangular boxes, and functions and procedures in rectangular boxes with duplicated vertical bars.

The initial job of module MAIN is to initialize variables and prepare all files required by PARUT. After this is completed, the user is required to respond to a prompt which enquires whether or not a data area requires analysis, and if not,

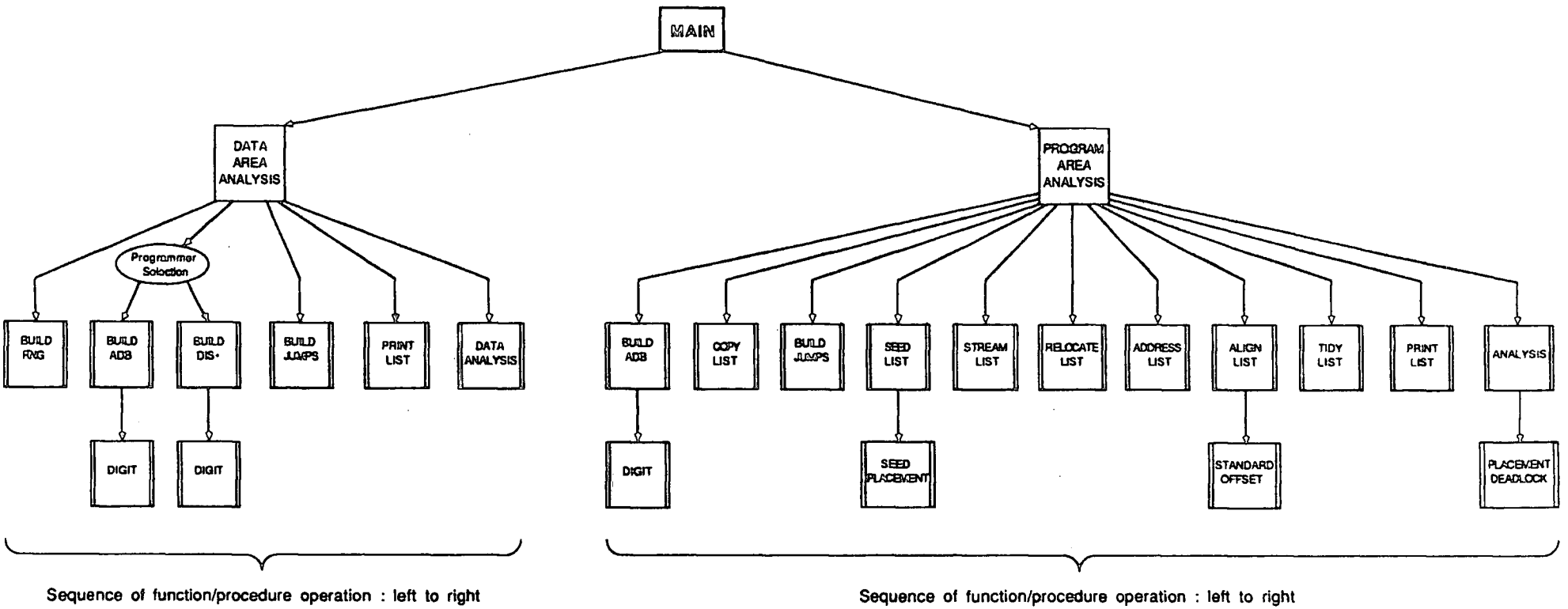


Figure 6.2. : Program 'MAIN' Call Chart

= Function or Procedure ; = Module Operation

then whether a program area requires analysis. If neither option is requested, the prompts are repeatedly presented until the user chooses an option. After completion of the requested analysis PARUT terminates activity and returns the user to the host environment.

Data and program code analysis implement a common multi-stage processing approach. The approach involves the following sequence of activity,

i) Generating a linked list to represent the machine code of the software under investigation. This includes the insertion of information within each record of the linked list detailing program flow associated with valid and invalid interpretation of the machine code.

ii) Apply fault tolerance by manipulating the linked list ensuring that the valid program flow is preserved (program code only).

iii) Produce a report documenting fault tolerant analysis of the machine code and, in the case of program code, detail the enhancement provided by the application of software implemented fault tolerant techniques.

6.5.1. Data Code Analysis

Analysis of data code involves four basic operations, each operation being contained within a function or procedure. Initially the user may request by prompt to construct a data structure either from actual data code (procedure BUILD_ADB), or from pseudo-random generated code (procedure BUILD_RNG). Actual data code was received in a UNIX 'DIS' format in an early version of the PARUT prototype, and for this the procedure BUILD_DIS was written. Now data code is received in a UNIX 'ADB' format and hence procedure BUILD_ADB is used, however, BUILD_DIS remains available for future use if required. After generating the data structure for the code, procedure BUILD_JUMPS is used to derive the program flow through the data area if it was incorrectly interpreted as program code. The penultimate operation of data area analysis is the output of the data code (primarily of use when the code

is generated by PARUT) using the procedure PRINT_LIST. Finally, the data code is analysed to determine the hazard of interpreting it as program code, and this is achieved by the procedure DATA_ANALYSIS.

6.5.2. Program Code Analysis

Program area analysis calls more procedures than data area analysis but retains the same basic approach. Initially a data structure is constructed by procedure BUILD_ADB to represent input program code. The user is then requested by a sequence of prompts to select a combination of software implemented fault tolerant techniques to be applied to the code including the technique proposed in Chapter 5.

Selection of any of the fault tolerant techniques offered to the user requires a duplicate copy of the linked list representing the target program code and this is provided by activating the procedure COPY_LIST. This copy is then processed by the procedure BUILD_JUMPS so that the necessary program flow information associated with both valid and invalid interpretation of the program code is incorporated. Then depending on the fault tolerant techniques chosen by the user, the procedures SEED_LIST (technique proposed in Chapter 5), STREAM_LIST (signature analysis), and RELOCATE_LIST (an alternative technique now discarded due to poor results) are executed. Other techniques can be added to those offered by PARUT, and would be included here in the structure of PARUT.

Once the selected techniques have been implemented on the copies of the linked list representing the original target program code, two 'housekeeping' operations are required. Firstly, procedure ALIGN_LIST is called which resets any disturbed absolute branches in the program code. In this manner PARUT does not compromise the transparent application feature of the software implemented fault tolerant technique (proposed in Chapter 5) upon the target software. Secondly, procedure TIDY_LIST is executed to remove any redundancy in the placed detection mechanisms. When the housekeeping is complete, the resultant code enhancement is output by procedure PRINT_LIST.

The operation of program code analysis terminates with a prompt to the user requesting a choice whether or not program flow analysis is required. If it is not required

then the action of this section of code is complete, otherwise the procedure ANALYSIS is activated. As a prerequisite to executing ANALYSIS the original program code data structure must be prepared for comparison with the enhanced program code version(s). This is achieved by the activation of the procedure BUILD_JUMPS. On returning from ANALYSIS this section of the PARUT code completes its operation. ANALYSIS reports instances of placement deadlock when the software implemented fault tolerant technique proposed in Chapter 5 is applied. A screen dump of the user interface for program code analysis is shown in Figure 6.3.

6.5.3. The 'Seeding' Algorithm

This section describes the algorithm used by PARUT to apply the fault tolerant technique proposed in Chapter 5. The algorithm is implemented by a procedure called SEED_LIST.

It is necessary before the algorithm is described to introduce some basic terminology and observations concerning the structure of invalid branches within machine code. The 'range' of an invalid branch describes the machine code locations lying between its generating location and target address. The range of an invalid branch has a 'level' which denotes how many target addresses of other invalid branches lie within its range. Within a section of machine code, invalid branches can 'group' incorporating features of four identified basic constructs. These constructs are shown in Figure 6.4. and are:

a) **Non-Intersecting Invalid Branches**

The generating locations and target addresses of the individual invalid branches range over independent areas of the machine code.

b) **Intersecting Invalid Branches**

The generating locations and target addresses of the individual invalid branches range over areas of the machine code that overlap. The range of one invalid branch contains the target address but not the generating location of the remaining invalid branch.

```
#Execution begins

 PARUT : TRANSIENT FAULT RECOVERY TOOL
 ====================================


 INFORMATION INPUT :- Please type appropriate response


 Data or Program Area (DATA/PROGRAM) ?
 PROGRAM
 Detection Mechanism Placement (YES/NO)?
 YES
 => Optimise Placement (YES/NO)?
 YES
 Boundary Relocation (YES/NO)?
 NO
 Signature Placement (YES/NO)?
 NO

 <<< Original Code (for comparison) being prepared >>>

 Analysis Required (YES/NO)?
 YES

#Execution terminated
```

Figure 6.3. : Screen Dump of PARUT User Interface

Figure 6.4. : Algorithmic Processing of Invalid Branches

c) **Coupled Invalid Branches**

Intersecting invalid branches except that ranges of both invalid branches contain the target address, but not the generating location, of the respective remaining invalid branch.

d) **Embedded Invalid Branches**

The generating location and target addresses of one invalid branch both lie within the range of the remaining invalid branch.

Of course there may be more complex situations of invalid branch interaction in the machine code under investigation, but such situations are constructs of the primitives listed above. The 'seeding' algorithm ensures that all invalid branches are resolved by 'seeding' except where placement deadlock is identified.

The Algorithm

Stage 1. Check whether or not there remain any unresolved invalid branches within the machine code. If not go to stage 8 of the algorithm.

Stage 2. 'Seeding' required. Investigate the machine code resolving invalid branches at level zero unless this is not the first pass of the code, in which case, increment the level to be investigated by 1.

Stage 3. Search the machine code until an unresolved invalid branch with the same level as that under investigation is found, or the end of the machine code is located. Searching commences initially from the start of the machine code. However, if an invalid branch has been resolved in the current code pass then the search commences at the location following the last address of the group in which that invalid branch was a member.

Stage 4. Resolve the invalid branch unless the end of the machine code was located in which case go to stage 6 of the algorithm.

Stage 5. Addresses are updated and valid program flow re-established for the machine code due to the insertion of a detection mechanism. Go to stage 3 of the algorithm.

Stage 6. Remove complex groups of invalid branches. If the present level of investigation is greater than zero then recursively apply the 'seeding' algorithm from stage 3 incrementing the level of investigation from zero to one lower than the present level.

Stage 7. If there remain unresolved invalid branches at the level of investigation after the code pass then return to stage 3 of the algorithm and start a new pass of the code, otherwise go to stage 1.

Stage 8. 'Seeding' complete.

The function of the 'seeding' algorithm is now demonstrated with the example of a complex invalid branch group shown in Figure 6.5. Noted below are the stages processed by the algorithm with status comments. The example should be examined in association with the 'seeding' algorithm.

Stage 1 : Unresolved invalid branches.

Stage 2 : 'Seeding' required. Level = 0.

Stage 3 : Start pass of linked list.

Stage 4 : Pass of linked list completed.

Stage 6 : No recursive call.

Stage 7 : No invalid branches at this level.

Stage 1 : Unresolved invalid branches.

Stage 2 : 'Seeding' required. Level = 1.

Stage 3 : Start pass of linked list.

Stage 4 : Invalid branch 'B' identified.

Stage 5 : Resolve coupled invalid branch.

Stage 3 : Complete pass of linked list.

Stage 4 : Pass of linked list completed.

Stage 6 : Recursive call to stage 3 with level = 0.

Stage 4 : No invalid branches at this level.

Stage 6 : Recursive call completed.

Stage 7 : Level 1 invalid branches remain unresolved.

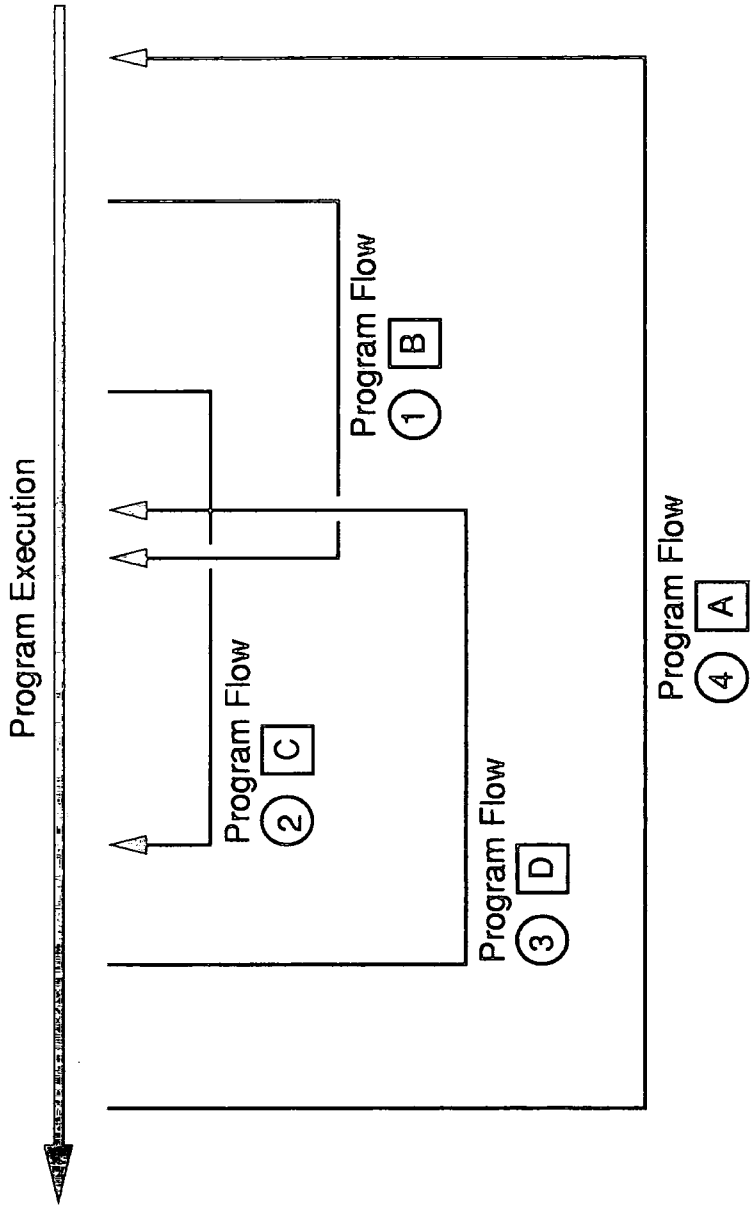


Figure 6.5. : A Complex Example of Algorithmic Processing

B = Invalid Branch Reference

(1) = Resolving Order of Algorithm

Stage 3 : Start pass of linked list.

Stage 4 : Invalid branch 'C' identified.

Stage 5 : Resolve coupled invalid branch.

Stage 3 : Complete pass of linked list.

Stage 4 : Pass of linked list completed.

Stage 6 : Recursive call to stage 3 with level = 0.

Stage 3 : Start pass of linked list.

Stage 4 : Invalid branch 'D' identified.

Stage 5 : Resolve embedded invalid branch.

Stage 3 : Continue pass of linked list.

Stage 4 : Invalid branch 'A' identified.

Stage 5 : Resolve non-intersecting invalid branch.

Stage 3 : Continue pass of linked list.

Stage 4 : Pass of linked list completed.

Stage 6 : Recursive call completed.

Stage 7 : Level 1 invalid branches resolved.

Stage 1 : All invalid branches resolved.

Stage 8 : 'Seeding' complete.

The procedural implementation of this algorithm is now briefly reviewed. Stages 1, 2, and 8 of the algorithm are implemented directly by procedure SEED_LIST, whilst the remaining stages are controlled by the called procedure SEED_PLACEMENT. Procedure SEED_PLACEMENT manages three procedures and recursive activation of itself. SEED_PLACEMENT initially executes procedure SEED_LOCATION to achieve stage 3 and 4 of the algorithm. This routine activates five other procedures. Initially JUMP_DIRECTION is used to determine the forward or backward nature of the invalid branch, then INTERVAL evaluates the level of the invalid branch and procedure TEST_SEED checks whether the invalid branch is already resolved by another detection mechanism placement. If the present invalid branch can be resolved then procedures SEED_DETAILS and PLACE_SEED construct and insert the detection mechanism into the machine code. SEED_DETAILS can place a default size or

optimum size detection mechanism, depending on user criteria passed by the root procedure MAIN. After executing SEED_LOCATION, SEED_PLACEMENT processes stage 5 of the 'seeding' algorithm by activating procedures ADDRESS_LIST and BUILD_JUMPS. These procedures are also directly used by the root procedure MAIN and are described in the following sections of this chapter. Finally, SEED_PLACEMENT implements repetitive and recursive calls to stages 3-7 of the 'seeding' algorithm in order to resolve complex invalid branch groups.

6.6. PARUT : A Review of the Prototype

The PARUT prototype successfully applies the software implemented fault tolerant technique proposed in Chapter 5. The 'seeding' algorithm employed by PARUT appears from experience to be efficient, but no quantitative assessment of its performance has been attempted. The algorithm is based on solving constructs of invalid branches: non-intersecting, intersecting, coupled, and embedded. The algorithm is also validated for an example of complex invalid branch group structure in machine code. In addition PARUT is designed to enable the simple inclusion of other software implemented fault tolerant techniques. In particular the prototype currently implements a simulation of the signature analysis technique. Further techniques can be included as required during any future development.

It is important that the operation of the prototype can be easily understood. A diagnostic facility is built into PARUT enabling the generation of a procedure call list referred to as TRACE_FILE. This list enables the operation of the utility to be monitored and hence aid comprehension of operation. Furthermore, programming language and style are adopted to facilitate understanding of the PARUT program. These qualities of the prototype have also proved valuable during the utility development, facilitating easy code manipulation without the disruption often associated with prototype development of similar sized programs.

A linked list is employed to represent the input machine code for processing by PARUT. This data structure is not dimensioned and does not itself restrict the size of machine code input. Equally, modules processing the linked list are designed not to impose a dimension restriction. However, there will be a constraint on the size

of the input code due to general limitations of the host system environment, e.g. a maximum size of CODE_FILE generation by UNIX 'adb'. Such restrictions are outside the scope of the PARUT development programme.

One of the most difficult objectives to achieve is the use of PARUT with a range of target processor types. Software manipulation required for the application of software implemented fault tolerance uses the pseudo-compiler action of procedure BUILD_JUMPS on the source code of the target processor. Such activity implies knowledge of the processor's instruction set, currently input to PARUT in MICRO_FILE. A robust version of PARUT should incorporate design features which facilitate a complete specification of a microprocessor type within MICRO_FILE to be processed by BUILD_JUMPS whose activity is independent of microprocessor architecture. Implementing such a robust specification is complex; therefore, for simplicity PARUT was developed to target only one processor type: the Motorola 68000 family. This family of microprocessors have a fixed size instruction set and extensions to the used instruction set are upwardly compatible. Hence, although only one processor type is made available by PARUT, the utility in reality can be used with a selection of processors within the Motorola 68000 family. This gives PARUT a base selection of target processors.

6.7. PARUT : Developing a Standard Programming Tool

The PARUT prototype extensively realizes its design and development objectives. It therefore appears feasible to further develop PARUT into a standard programming tool. Such a tool is valuable when implementing and assessing fault tolerance associated with the characteristics of erroneous behaviour described in Chapter 3.

A standard programming tool based on the prototype PARUT should adopt the following recommended enhancements. Firstly, the range of target processors should be extended. This is possibly the most complex modification of PARUT involving the integrated development of a robust BUILD_JUMPS module and general purpose format for MICRO_FILE. Secondly, PARUT should be extended to implement (rather than simulate) other software implemented fault tolerant techniques. The program structure of the PARUT program has been demonstrated to provide easy inclusion

of new techniques. Thirdly, the analysis techniques used to assess the effectiveness of the fault tolerant techniques implemented could, in addition to static analysis, provide dynamic analysis via emulation/ simulation.

As a standard programming tool PARUT might be incorporated into a compiler. This would remove the necessity of generating and processing CODE_FILE because all the required information on the target software is inherently available from the translation process of the compiler.

6.8. Summary

This chapter describes the design and development of the prototype programming tool PARUT. Design objectives are successfully attained. In particular a selection of software implemented fault tolerant techniques, including that proposed in Chapter 5, are facilitated for a variety of target processors. This is achieved without undue restrictions on the size of the target software. Additionally the fault tolerance of the software can be assessed in respect of the hazard of erroneous microprocessor behaviour modelled in Chapter 3.

The structured programming language Pascal is used in conjunction with 'good' programming practice to generate readable code and hence ease comprehension and modification. A diagnostic facility which monitors procedure access by PARUT operation is also provided to aid understanding of the utility function. All these design features have proved valuable during the development of the PARUT prototype.

The success of PARUT leads to the proposal that further development be initiated in order to produce a standard programming tool. Enhancements to PARUT for this purpose are outlined. The post-programming nature of the software implemented fault tolerant techniques applied by the PARUT function suggests its possible inclusion within a compiler, providing an additional code enhancement stage at the end of the translation process.

Chapter 7

ASSESSING FAULT TOLERANCE

CHAPTER SEVEN

ASSESSING FAULT TOLERANCE

7.1. Introduction

The effectiveness of software implemented fault tolerance in a microprocessor system is difficult to assess. Microprocessors and their memories are VLSI devices which have a huge number of potential logical fault sites. In addition these faults will not always be activated because of time dependent circuit operation. This chapter reports two fault insertion experiments initiated in order to investigate temporary faults within a microprocessor system implementing the software based fault tolerance technique proposed in Chapter 5.

The fault insertion experiments investigate the effect of single-bit and multiple-bit faults on program behaviour. The faults are inserted into memory and the program counter, and the response of the microprocessor system tracked instruction by instruction.

The first set of experiments involve injecting single-bit faults into a microprocessor system. Five classes of single-bit fault are injected in order to model faults at various locations in the microprocessor and memory. The faults include line-errors on the address and data bus during instruction and data cycles, and program counter faults. The response of the microprocessor system once the faults have been injected is monitored. A detailed analysis of the system response gives an important insight into the character and nature of erroneous microprocessor behaviour.

Within microprocessor based systems faults are often observed to affect multiple-bit as well as single-bit locations. This occurrence is investigated by the second set of fault injection experiment. The fault class selected for multiple-bit fault injection is program counter corruption. This fault is selected because it is observed in many of the single-bit fault injection experiments, and it is relatively simple to inject. The fault response of three microprocessor systems (an 8, 16, and 32-bit machine), is analysed.

Finally, the chapter concludes with a summary of experimental observations relating to the effectiveness of the software detection mechanisms and the Access Guardian function. In particular, detection latency and the hazard of re-synchronization are discussed.

7.2. Assessing the Fault Tolerance of a Microprocessor System

7.2.1. Assessment Parameters

Techniques providing tolerance of temporary faults in VLSI devices are difficult to assess. This is particularly true for microprocessor systems. In order to assess the effectiveness of a fault tolerant technique it is necessary to evaluate several performance parameters. Two performance parameters are particularly important: detection latency, and error coverage. Error coverage is the percentage of error conditions that can be detected by the technique, and the time taken between the activation of a fault as an error and its detection is referred to as its detection latency. It is also important to quantify the performance overhead imposed by the technique. The overhead comprises processing degradation and additional hardware requirement.

7.2.2. Parameter Evaluation

Analytic assessment of fault tolerant microprocessors is a difficult task, because in most applications it is impossible to determine appropriate error classes and the distribution of errors amongst these classes. Although models have been developed to investigate the effect of temporary hardware faults on executing microprocessor systems, their analysis is limited by assumptions and imposed restrictions. Mahmood & McCluskey [1985], and Namjoo [1982] have modelled the error coverage of signature analysis techniques but their investigation was limited to the effect of control flow errors, and their estimates proved slightly optimistic compared with actual results presented by Schuette & Shen [1986]. Nevertheless, the model did give a valuable indication of the effectiveness of signature analysis.

Experimental evaluation by fault injection into actual hardware in many cases is the only way to estimate fault tolerance effectiveness successfully. In such experiments

the selection of the fault injection method is crucial. Ideally a fault should be capable of injection at random and specific VLSI device locations, and at a certain point in time for a controlled period. Executing software is dynamic and temporary faults can have different outcomes depending on the activation of circuitry by microprocessing within the processor. This is referred to as the instruction sequence dependency. Higher processor workloads, in a multi-tasking environment, may also increase the probability of activating a fault.

Temporary faults that occur in microprocessor devices are difficult to mimic in the laboratory. Initially fault injection experiments, see Table 2.3., inserted faults via the hardware pins of a device. More recently Chillarege & Bowen [1989] inserted faults into a microprocessors memory. Whilst these faults model internal faults, faults are not actually injected within the device. Although there are methods of injecting internal faults to a microprocessor, notably Damm [1988] by power line fluctuations, and Gunneflo et al [1989] by ion radiation, these methods generate multiple faults at uncontrolled locations.

Controlled fault generation can be provided by using microprocessor emulators and simulators, see Table 2.2., but analysis of the microprocessor response may be limited by the tool's sophistication. Armstrong & Devlin [1981] suggest that gate-level microprocessor simulators are prohibitively expensive for fault injection experiments. Therefore, they used a microprocessor simulator based on a functional model [Li et al, 1984]. More sophisticated emulators have become available to researchers, whereby gate-level descriptions are incorporated into functional models. Czeck & Siewiorek [1990] recently used such a sophisticated simulator. However, it must be realized that as the simulators increase in complexity so they become increasingly susceptible themselves to development errors. A simulator was not available to the author of this thesis so an alternative method of assessing fault tolerance is utilized.

As mentioned above, an accepted method used by many researchers to obtain assessments of fault tolerant techniques is fault injection. The method is popular because it gives actual error coverage analysis for the injected faults. A limitation of the method is the validity of the result for the whole device. Locations for fault

injection are usually chosen by the experiment designer following some predefined selection criteria. The representivity of these points for the remainder of the device should always be critically examined. This method appeared to be the best approach to analyse the software based fault tolerant technique proposed in Chapter 5.

7.2.3. Internal Microprocessor Faults

Faults are selected for insertion in order to model actual faults in a microprocessor system. The inserted faults are only valid when modelling processor and memory faults, faults in the peripheral devices are not accurately modelled. Before comparing inserted and actual faults, it is necessary to describe the basic structure of the tested microprocessor system.

The microprocessor system under test is considered as the integration of a data path (consisting of an ALU and data registers), and a control path (consisting of an opcode decoder, controller, and program control unit). The program control unit (PCU) contains bus interface circuitry (BIC), program counter (PC), instruction prefetch queue (IPQ), and a controller. The PCU is responsible for address calculations, and it is assumed to be responsible for the reading and writing of opcodes and operands. Interrupt handling and bus arbitration circuitry will not be considered because the inserted faults will not closely model faults within these units. A general microprocessor topology is shown in Figure 7.1. The microprocessor system under test does not have the following features: co-processor, memory cache, instruction pipeline (beyond the prefetch queue), and multiplexed busses.

7.2.4. Assessment Dependence on Application Software

The effectiveness of software based fault tolerant techniques is difficult to access. Techniques such as recovery blocks, signature analysis, and the technique proposed in Chapter 5 are all dependent on the application program for their performance. It is therefore important to select a representative application program when assessing the effectiveness of a fault tolerant technique. Czeck & Siewiorek [1990] and Schuette & Shen [1986] each choose a different collection of application programs

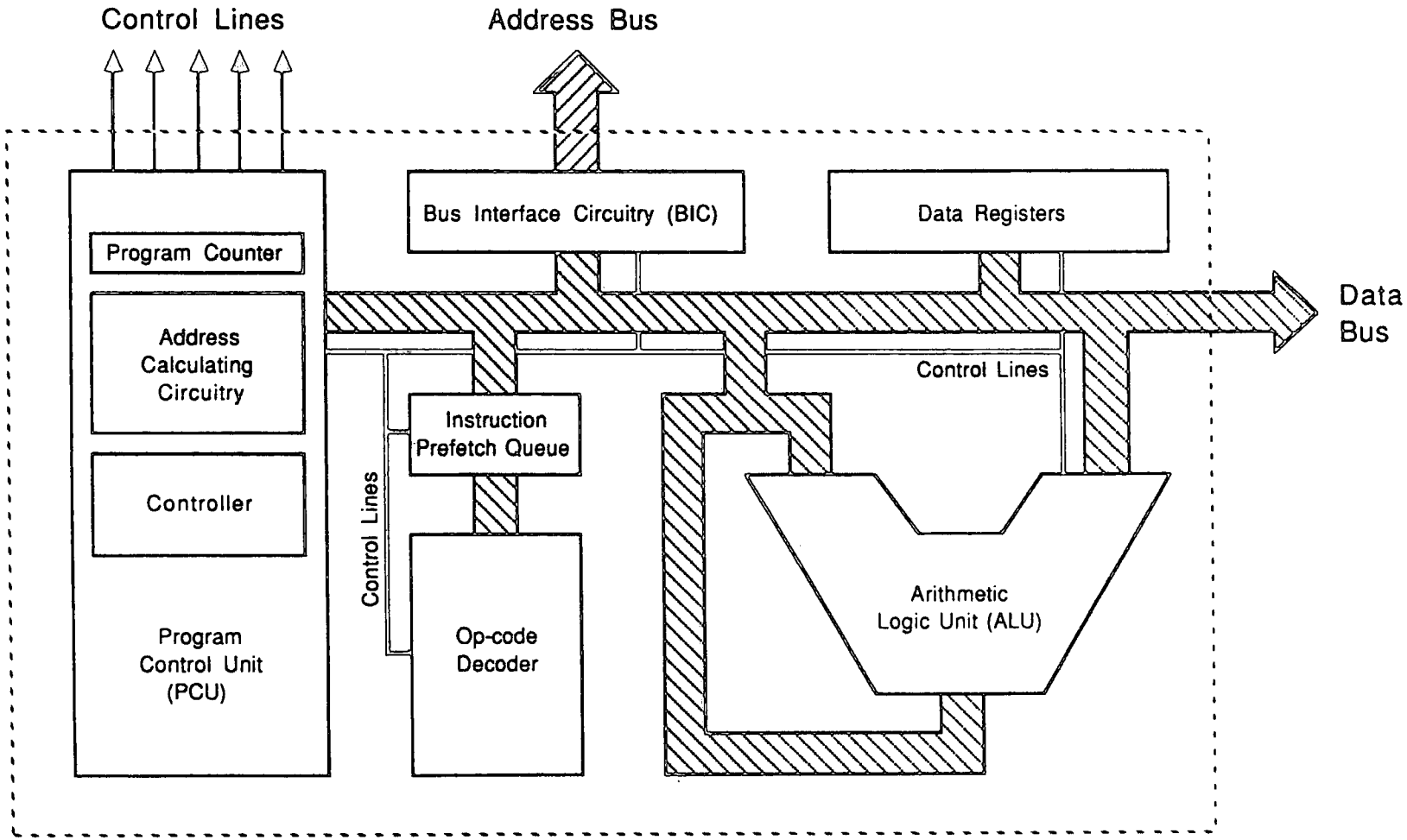


Figure 7.1. : Basic Microprocessor Topology

in order to provide an assessment benchmark. However, particular applications for implementation can have quite different characteristics from any of the benchmark software. Hence the value of a benchmark program is limited to giving an indication of the effectiveness of a fault tolerant technique implemented on a similar application program.

7.2.5. Behavioural Observations

Recently reported fault injection experiments (see Table 2.2.) measure the detection latency of fault tolerant techniques. Such experiments do not enable the mechanism of spawning errors, at a functional level, to be observed. The fault injection experiments described within this chapter involve tracing the instruction sequence of microprocessor operation from the activation of the fault to its detection or deactivation. These experiments provide an interesting insight into the mechanisms, modelled in Chapter 3, of processing failures induced by temporary faults.

7.3. Single-Bit Fault Injection Experiment

7.3.1. Fault Injection Experiment

The experimental set-up of the fault injection programme is shown in Figure 7.2. Within the Engineering Department at the University of Durham there is a Vittese 5 computer system based on the Motorola 68020 microprocessor which services many devices, including terminals independently connected through a dedicated Motorola 68000 microprocessor system, in a multi-user environment. Programs written for implementation on a dedicated Motorola 68000 system are coded on the Vittese in Assembler before being assembled. The object code can then be down-loaded onto an MC68000 microprocessor system if one is attached to the terminal in use. The MC68000 microprocessor system used was developed by the Microprocessor Centre at the University of Durham, and has a special 'monitor' program which provides, amongst other facilities,

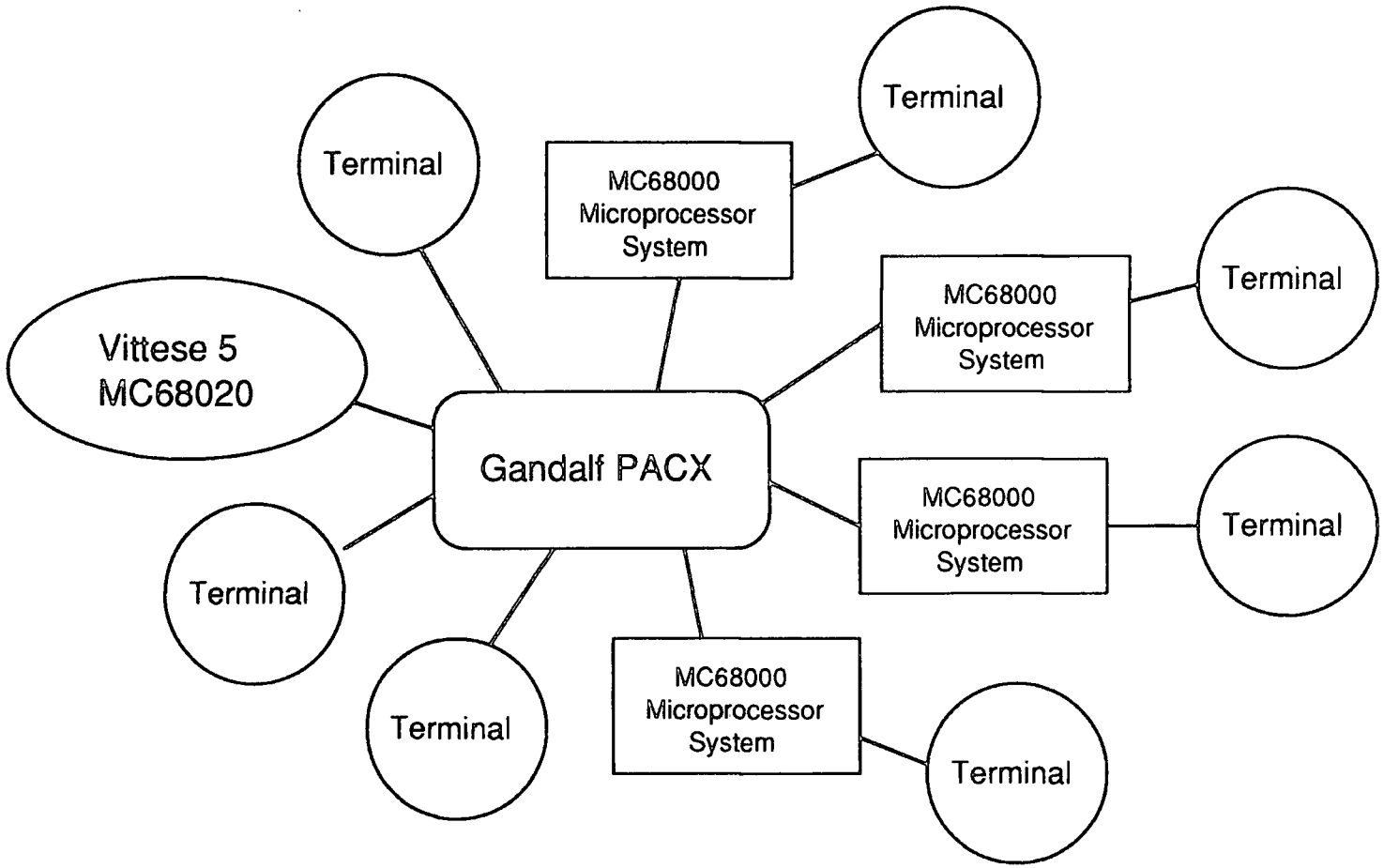


Figure 7.2. : Microprocessor Laboratory System Topology

- read/ write memory locations.
- read/ write register contents.
- trace execution (instruction by instruction).

The fault injection programme implemented involves injecting faults into the microprocessor memory or program counter. These faults are activated by the application program's execution, and their effects are monitored by using the trace facility and interrogating the register contents to ascertain the processor's status.

7.3.2. Microprocessor Application Under Investigation

For the purpose of this section, a single microprocessor application is analysed. The system chosen has many typical characteristics of an industrial microprocessor based controller such as the monitoring and control of equipment to perform an on-going task or process.

The application system monitors the water level in two connected reservoirs, one higher than the other. If the higher reservoir level falls beneath a minimum marker (solenoid float) then water is pumped from the lower to the higher reservoir. Similarly, if the higher reservoir level goes above a maximum marker (solenoid float) then water is drained from the higher to the lower reservoir.

The controller is based on a Motorola 68000 microprocessor operating at 8MHz although a lower operating frequency would be acceptable for this application. The microprocessor executes software stored within 64 KBytes of memory. The actual program size is initially 381 Bytes. The application program is processed by the PARUT tool, described in Chapter 6, in order to strategically place software detection mechanisms within the code. These mechanisms, designed to provide fault tolerance, for the application program under investigation required an additional 108 Bytes of memory. An annotated listing of the original application program, and the same program after processing by PARUT can be found in Appendix D respectively.

The Motorola 68000 microprocessor system has several detection mechanisms; bus errors (access to unavailable address space) are detected by logic external to the processor whilst address errors (invalid specification of memory locations , e.g. odd

byte address exception) and illegal opcodes are inherently detected by the processor. Collectively these mechanisms shall be referred to as the MC68000 detection facility.

The bus and address errors within the MC68000 detection facility duplicate the function of the Access Guardian proposed in Chapter 5. Therefore an Access Guardian unit is not attached to the microprocessor system. The collected results from the fault injection experiments are processed in order to de-couple the Access Guardian function from the MC68000 detection facility.

7.3.3. Programme of Injected Faults

The fault insertion programme was based on that used by Schuette & Shen [1986]. They injected faults by temporarily altering pin logic values on an embedded Motorola 68000 microprocessor. The faults in this insertion programme are injected by corrupting the microprocessor memory. The experiment models five classes of fault within the microprocessor system as described below.

A. Instruction Cycle : Data Bus Faults

These faults are inserted by corrupting bit positions of an opcode stored in memory. The fault appears on the data bus when the instruction opcode is read. This class of fault models the following situations.

- Memory degradation or data bus line-errors external to the microprocessor.
- Errors in the bus interface circuitry or internal data bus line-errors.
- Errors in the Opcode Decoder.
- Program Counter faults or Address Calculating Circuitry errors as a result of either determining an incorrect branch address (special MC68000 case where displacement is held in the opcode), or corrupted opcode, or opcode decoder error leads to the wrong number of operands being read and hence an incorrect location is accessed for the next opcode.

B. Data Cycle : Data Bus Faults

These faults are inserted by corrupting bit positions of an operand in memory, the fault appearing on the data bus when the operand is accessed. This class of fault models the following situations.

- Memory degradation or data bus line-errors external to the microprocessor.
- Errors in the bus interface circuitry or internal data bus line-errors.
- Faults in the data registers when operands are moved by them.
- Arithmetic Logic Unit (ALU) errors if the operands are processed.
- Program Counter faults or Address Calculating Circuitry errors if the operand is used in determining a branch address.

C. Instruction Cycle : Address Bus Faults

These faults are inserted by replacing an opcode in memory with data from another location in the address space. The fault is activated when the opcode is accessed and an alternative opcode value is put on the data bus. This class of fault models the following situations.

- Memory degradation or data bus line-errors external to the microprocessor.
- Errors in the bus interface circuitry or internal data bus line-errors.
- Faults in the stack pointer (when retrieving the next opcode location), program counter faults, or errors in the Address Calculating Circuitry.
- Multiple faults in the Opcode Decoder which cause severe malfunction and have an effect similar to multiple line-errors on either the internal or external data bus, and burst faults in memory.

D. Data Cycle : Address Bus Faults

These faults are inserted by replacing an operand in memory with data from another location in the address space. The fault is activated when the operand is accessed and an alternative operand value is put on the data bus. This class of fault models the following situations.

- Memory degradation or data bus line-errors external to the microprocessor.
- Errors in the bus interface circuitry or internal data bus line-errors.
- Errors in the Address Calculating Circuitry.
- Alternatively this fault class can mimic multiple faults in the data register, or ALU malfunction, or multiple line-errors on the internal or external data bus, or burst faults in memory.

E. Program Counter Faults

These faults are inserted by corrupting the contents of the program counter using the status facility in the MC68000 board monitor program. The fault becomes active when the next opcode is accessed and processing is forced to deviate from its intended path. This class of fault models the following situations.

- Line-errors on the internal address bus.
- Opcode Decoder faults initiating a branch.
- Corruption of the program counter, errors in the Address Calculating Circuitry, or stack pointer faults which lead to an incorrect branch.

Review

A summary of the injected fault programme and modelled fault situations can be found in Table 7.1. The injected faults are all single-bit; multiple bit faults were not injected in this experiment. The modelled faults are single-bit, or simple errors, except where stated.

7.3.4. Selected Single-Bit Faults

The faults injected for the instruction and data cycle address and data bus experiments, and the corrupted program counter experiment are single-bit corruptions. Bit faults on the address bus and program counter affect address bit positions 1, 4, 8, and 12. Bit fault positions on the data bus are 0, 7, 8, and 15. These bit positions are used by Schuette & Shen [1986] and Damm [1988] in their fault injection

Fault Modelled		Fault Class Injected				
		A	B	C	D	E
Memory Bit Faults		X	X	X[6]	X[6]	
Memory Select Circuitry Error				X	X	
Line Errors :	Internal Data Bus	X	X	X[6]	X[6]	
	External Data Bus	X	X	X[6]	X[6]	
	Internal Address Bus			X	X	X
	External Address Bus			X	X	
Bus Interface Circuitry Errors		X	X	X	X	
Faults :	Data Registers		X		X[6]	
	ALU		X		X[5]	
	Opcode Decoder	X		X[5]		X
Program Counter Faults		X[2,3]	X[1]	X		X
Address Calculating Circuitry Errors		X[2,3]	X[1]	X	X	X
Stack Pointer Faults				X[4]		X

Fault Class Injected:
A: Instruction Cycle: Data Bus Faults
B: Data Cycle: Data Bus Faults
C: Instruction Cycle: Address Bus Faults
D: Data Cycle: Address Bus Faults
E: Program Counter Faults

Notes:
(1) Only if the operand is used in determining a branch address.
(2) Special MC68000 case (1): displacement held in opcode.
(3) Corruption of an opcode or a fault in the Opcode Decoder can result in the wrong number of operands being read for an instruction and hence a program counter or address calculating circuitry error when accessing the next opcode.
(4) When retrieving next opcode address from stack.
(5) Multiple faults causing severe malfunction.
(6) Multiple faults causing severe mis-interpretation of opcode/operand.

Table 7.1. : Single-Bit Fault Injection Programme

experiments. The application program resides in low memory and hence address bus faults on line 12 are typically detected by the Access Guardian function.

In total 2136 faults were injected during the single-bit fault experiment. These faults disrupted program and data flow. A feature of the fault injection programme is that results are dependent on the instruction sequence and not the instruction mix: actual execution paths are traced instruction by instruction. This is important because faults can have different effects when the the microprocessor system is in various run-time conditions.

Each instruction cycle fault class had 472 faults injected, as did the program counter fault experiment. The data cycle fault classes comprise of 360 faults each.

7.3.5. Decoupling the Microprocessor Detection Mechanisms

The microprocessor system under investigation has two sources of detection mechanisms: those implemented by the software based fault tolerant techniques proposed in Chapter 5, and those inherently present in the embedded MC68000 microprocessor.

Let the sample space F contain all the faults injected into the microprocessor system. Let the sets M and P be the faults covered by the MC68000 detection facility and the software detection mechanisms planted in the application program respectively. Now,

$$(M \cap P) = \emptyset, \tag{7.1.}$$

but unfortunately M and P do not describe the whole fault set F.

$$(M \cup P) \subset F. \tag{7.2.}$$

This is because the errors generated by some faults cause erroneous execution to re-synchronize and hence avoid detection.

Let R be the set of faults that generate a re-synchronized outcome, and therefore the injected fault outcomes can be described as follows.

$$(M \cup P) \cup R = F, \tag{7.3.}$$

$$(M \cup P) \cap R = \emptyset. \tag{7.4.}$$

The MC68000 detection facility is observed to detect fault outcomes which would otherwise have been diagnosed by an Access Guardian. For the purpose of analysis it is useful to de-couple the Access Guardian function from the inherent MC68000 detection mechanisms. Let set A contain all the faults that are detectable by the Access Guardian function. Then

$$(M \cap A) = A, \tag{7.5.}$$

$$(M \cup A) = M. \tag{7.6.}$$

The faults detected by the MC68000 detection facility without the Access Guardian function is therefore given by $(M \cap \bar{A})$.

7.3.6. Performance Evaluation

The detection mechanisms provided fault tolerance for approximately 60% of the faults injected into the system. The outcome of the faults no detected is observed as re-synchronization. The mean latency to either detection or re-synchronization is 1.2 processed instructions. The implications of this latency is discussed below. The overhead associated with the implementation of the software based fault tolerant technique have been estimated in Chapter 5. The hardware overhead attributed to the Access Guardian is approximately 5% of the MC68000 transistor count. The additional memory required by the fault tolerant software is approximately 30% of the original application program size.

A summary of the processing outcome after each single-bit fault is injected is shown in Table 7.2. The processing outcomes are classified as re-synchronization,

Injected Fault Class	Re-synchronisation		Detected by			Total
	with stack corruption	without stack corruption	Software Mechanism	Access Guardian Function	MC68000 excluding Access Guardian Function	
A	2	133	121	182	31	472
B	2	210	2	146	0	360
C	5	132	91	190	54	472
D	10	198	6	146	0	360
E	36	145	121	150	20	472
Total	55	818	341	814	105	2136

Table 7.2. : Fault Injection Outcomes

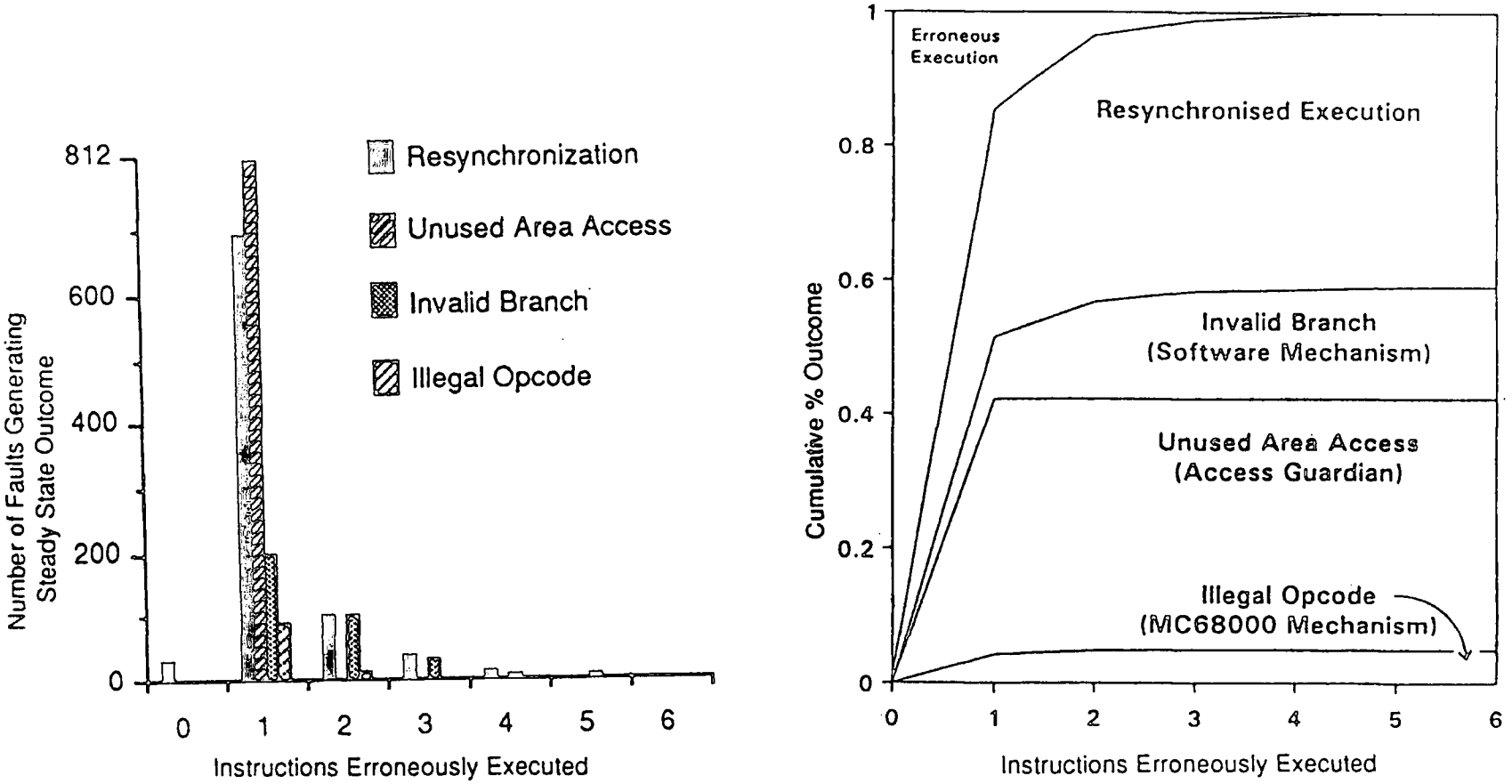
detection by a software mechanism, detection by the MC68000 microprocessor excluding the Access Guardian function, and finally, detection by the Access Guardian function. The processing response en route to the monitored outcomes is detailed in Table 7.3. and shown in Figure 7.3.

Re-synchronization involves the program flow, which has already diverged from the specified control path, rejoining the specified control path. Over 40% of the injected faults lead to re-synchronization. This is not surprising, as a significant proportion of the faults injected during the experiment corrupted the instruction without initially affecting the program counter, but corrupted the program counter following the completed execution of the first instruction because the wrong number of operands were interpreted as belonging to the initial opcode. This scenario initiates irregular processing through the coded area before processing once more aligns itself with the original program flow. Gunneflo et al [1989] recorded 5% of his injected faults leading to re-synchronization, the lower value of this figure can be attributed to the nature of his fault experiments. Arlat et al [1989] and Chillarege & Bowen [1989] report 46% and 42% of their fault injections leading to undiagnosed errors which did not prevent continued processing although the function may have been slightly corrupted. These faults together with similar faults, referred to as overwritten faults and collated by Czeck & Siewiorek [1990], (59% [Schuette & Shen, 1986], 60-70% [Czeck & Siewiorek, 1990], and 77% [Choi et al, 1989]), may to some degree be due to the re-synchronization phenomena.

It is clear that re-synchronization is an important class of fault outcome. In the experiment re-synchronization occurred within five erroneously processed instructions, and had a mean occurrence latency of 1.2 instructions. It is interesting to note that 29 of the injected faults classified as re-synchronization, had no effect on execution - the induced erroneous behaviour being totally benign as far as the system status is concerned. For example, in the data cycle with address bus fault, the incorrectly accessed operand address may contain the same operand value as the correct operand address. These occurrences are shown in Table 7.3. where re-synchronization is labelled as occurring after no instructions are processed.

Injected Fault Class	Re-synchronise (not detected)						Detected by Software Mechanism								Access Guardian Detection Function				MC68000 Detection Excluding Access Guardian Function					Total
	0	1	2	3	4	5	0	1	2	3	4	5	6	7	0	1	2	3	0	1	2	3	4	
A	12	78	35	9	1	0	0	70	33	13	1	5	2	0	0	180	2	0	0	21	9	1	0	472
B	17	194	1	0	0	0	0	0	1	0	1	0	0	0	0	146	0	0	0	0	0	0	0	360
C	0	107	8	13	9	0	0	57	30	4	0	0	0	0	0	190	0	0	0	52	1	1	0	472
D	0	202	6	0	0	0	0	1	4	1	0	0	0	0	0	146	0	0	0	0	0	0	0	360
E	0	116	50	14	1	0	0	68	32	12	6	3	0	0	0	150	0	0	0	17	3	0	0	472
Total	29	697	100	36	11	0	0	196	100	30	8	8	2	0	0	812	2	0	0	90	13	2	0	2136

Table 7.3. : Nature of Fault Injection Outcomes



(a) Individual Instruction Outcomes

(b) Cumulative Instruction Outcomes

Figure 7.3. Fault Injection Outcomes

A particular hazard associated with re-synchronization is that from a user perspective there is a small and perhaps un-noticeable processing glitch which can involve corruption of the microprocessor stack or stack pointer. Approximately 3% of the injected faults re-synchronize with a corrupted stack. A similar result is reported by Damm [1988] who diagnosed 3.6% of his injected faults to cause this error. This hazard may prove critical at a much later processing stage when the return from a subroutine or other stack access occurs. This phenomena of a sleeping fault, called a potential hazard by Chillarege & Bowen [1989], being awakened at some future period could explain the system crash data presented by Czeck & Siewiorek [1990] where high corruption of system integrity is experienced.

Those faults injected to disrupt the data cycle have a 56% probability (approximately) of generating re-synchronization, which is twice the expectation for instruction cycle faults. This is not an unexpected result because corrupted operands will typically alter the result of the function but not the function itself whilst corrupted opcodes will alter the function and interpretation of operands. It is interesting to note that the program counter faults, which are intuitively more allied to instruction cycle faults, generate re-synchronization for 30% of their outcomes which is very similar to the data cycle fault experiment.

In the experiment non-re-synchronized erroneous execution was detected, by either a software detection mechanism, Access Guardian function, or a non-Access Guardian function of the MC68000 detection facility, with a mean latency of 1.2 instructions. This is extremely rapid and highly desirable for reliable systems. The faults trapped with greatest latency (6 instructions) were detected by the software detection mechanism.

Just over one quarter of the faults which do not re-synchronize, are caught by the software detection mechanisms. Only 8, or 1%, of the data cycle faults are detected in this way, compared with 20% of the instruction cycle and program counter faults. The software detection mechanisms have caught erroneous execution as late as the sixth processed instruction, and have a mean detection latency of 1.6 instructions in the experiment.

The Access Guardian function was very successful in detecting injected faults. It detected 40% of the faults inserted with a mean detection latency of one processed instruction. The experimental result reported here compares favourably with other documented versions of this detection function; 60% [Gunneflo et al, 1989] where the used memory filled 12% of the Motorola 6809 microprocessor address space, and 58% [Schmid et al, 1982] where the used area filled 90% of the Zilog Z80 microprocessor address space. As noted by Gunneflo et al [1989], the effectiveness of this detection mechanism will increase as the percentage of used memory in the microprocessor address space decreases.

The remaining injected faults were detected by the illegal opcode facility of the MC68000 microprocessor. These accounted for almost 5% of the detected faults. Early microprocessor designs did not incorporate this facility and under these fault injection experiments would have a reduced reliability. Schmid et al [1982] and Gunneflo et al [1989] attached an illegal opcode detector to their respective Z80 and MC6809 microprocessor systems; the facility detected approximately 35% and 30% of the injected faults. Most modern microprocessor designs incorporate this detection capability. The effectiveness of this mechanism is dependent on the number of illegal opcodes in the instruction (opcode) map, and the data diversity within the microprocessor used memory which is application dependent. It is therefore difficult to quantify the usefulness of this utility, but it can considerably improve a microprocessor system's reliability.

7.4. Multiple-Bit Fault Emulation Experiments

7.4.1. Emulation and Fault Investigation

Emulators are software tools that mimic the register action of a target microprocessor. As such the injection of a fault will not be as accurately modelled as in a simulator which models the functional/gate activity of a microprocessor. However, emulators, unlike simulators, are commonly available and inexpensive. Indeed many modern microprocessor systems are provided with an emulator within a debugging facility.

The register selected for fault investigation is the program counter. Other register faults would only investigate data type errors, whilst program counter faults are indicative of instruction type faults. Gunneflo et al [1989], who injected faults internally at random via ion radiation, found that 77% of faults were instruction type. Other fault injection experiments support this finding: 77% [Schuette & Shen, 1986] and 60% (experienced in the single-bit experiment documented within this chapter). It therefore seems reasonable to conduct experiments that investigate instruction type faults.

7.4.2. Microprocessor Applications Under Investigation

Three application programs have been selected for the multiple-bit fault injection experiment. The first program ('A'), is the same program used for the single-bit fault injection experiment. That is, a slurry pump control application involving the monitoring and control of a reservoir system. Program A is written in assembler for the Motorola 68000 microprocessor. The second program ('B'), is written in assembler for a Motorola 68(7)05 microprocessor based system and is concerned primarily with data movement using the processor input/output ports. The third program ('C'), unlike the other programs, is not an application program but rather a section of high level code written in C. The purpose of this program is to investigate the hidden hazards that can be generated when high level language programs are translated to machine code. Program C is translated into machine code for the Intel 80386 microprocessor.

The three programs were selected to provide a diverse variety of application processors types and sizes; the Motorola 68(7)05, Motorola 68000, and Intel 80386 are 8, 16, and 32-bit machines respectively. As for the single-bit fault injection experiment, the programs are prepared by applying the software based fault tolerant technique proposed in Chapter 5. Annotated assembler/ machine code listings of the original application programs and the programs with strategically placed software detection mechanisms are shown in Appendix D.

7.4.3. Programme of Emulated Faults

The faults injected into the program counter are single and multiple-bit corruptions. All possible program counter corruption patterns representing execution in the used area of the microprocessor address space are analysed through fault injection. The remaining program counter corruption patterns generate detection by the Access Guardian and are evaluated. Hence this class of fault is comprehensively analysed. High order bit faults in the program counter typically lead to detection by the Access Guardian feature because the application program resides in low memory.

The programs A, B, and C are investigated to assess their respective fault tolerance with and without the software based fault tolerant technique proposed in Chapter 5. Each program is evaluated :-

Version 1 : without the software technique applied,

Version 2 : with the software technique applied (default detection mechanism size),

Version 3 : with the software technique applied (optimum detection mechanism size),

Program A : *{68000}*

The size of this program is 308 bytes, increasing to 500 and 416 bytes when default and optimum size software detection mechanisms are inserted respectively. Faults are emulated to analyse the microprocessor response to even byte program counter corruption covering every location in the program for each of the three program versions; a total of 612 fault runs. Odd byte program counter corruptions are detected automatically by the inherent MC68000 detection facility.

Program B : *{68(7)05}*

The size of this program is 53 bytes, increasing to 86 and 80 bytes when default and optimum size software detection mechanisms are inserted respectively. Faults are emulated to analyse the microprocessor response to program counter corruption covering every location in the program for each of the three program versions; a total of 219 fault runs.

Program C : {80386}

The size of this program is 293 bytes, increasing to 365 and 323 bytes when default and optimum size software detection mechanisms are inserted respectively. Faults are emulated to analyse the microprocessor response to program counter corruption covering every location in the program for each of the three program versions; a total of 981 fault runs.

7.4.4. Behavioural Analysis

The performance evaluation in the preceding section, concerning the fault injection experiment, and other fault tolerance evaluations (see Tables 2.2. and 2.3.) involving fault injection, simulation, or emulation, provide static analysis. They do not investigate the processing behaviour associated with the latency of the spawning errors generated by the injected fault, and hence cannot identify dangers or assets of the techniques under evaluation.

The emulation experiment reported here involves tracing erroneous execution instruction by instruction. In this way, the character of erroneous microprocessor behaviour can be investigated. This study validates the erroneous microprocessor behaviour model presented in Chapter 3, demonstrates the effectiveness of the software based fault tolerant technique proposed in Chapter 5, and re-iterates the importance of the re-synchronization phenomenon.

7.4.5. Identified Phases of Erroneous Execution

The erroneous microprocessor behaviour model assumes two phases of erroneous execution: that following an Initial Erroneous Jump (IEJ), and that following a Subsequent Erroneous Jump (SEJ). The fault programme primarily investigates the SEJ phase, but the results can be extended to investigate the IEJ phase under the assumption that an Access Guardian is embedded within the microprocessor system being evaluated. Execution within each phase can generate either a restart, unspecified jump, return, or stop/wait outcome as described in Chapter 3. Restart outcomes signify detection of erroneous execution, whilst stop/wait outcomes signify a

cessation of execution. Only unspecified jump and return outcomes lead to another SEJ phase of erroneous execution.

The fault emulation results for program A, shown in Table 7.4., are analysed in detail in order to validate the erroneous microprocessor behaviour model. Erroneous execution can be detected in the MC68000 microprocessor system by either the Access Guardian, the inherent MC68000 detection mechanisms, or the software detection mechanisms. The Access Guardian detects access to the unused address space. Inherent MC68000 processor detection mechanisms include the odd byte address exception for the program counter, and the illegal opcode exception. The software detection mechanisms are designed to detect SEJs.

7.4.5.1. The Initial Erroneous Jump Phase

The purpose of Figure 7.4. is to show the behavioural phase of program execution following an Initial Erroneous Jump (IEJ). The IEJ is generated by corrupting the program counter. Some IEJ destinations are detected immediately, such as target locations in the unused area and odd byte addresses, and these are represented by the ordinate intercept in Figure 7.4. During erroneous execution such detection coerces the outcome of an instruction to a restart. The ordinate intercept is lower for version 2 because the injected software mechanisms increase the used memory requirement, which reduces the initial effectiveness of the Access Guardian. The odd byte address exception facility in the MC68000 microprocessor detects all program counter corruptions with an odd byte address, and hence will always have the same detection capability.

During the IEJ phase, detection is provided by the software detection mechanisms or instructions generating conditions that are detected by the Access Guardian or an inherent MC68000 detection mechanism. Version 1 does not have any inserted software detection mechanism, whilst version 2 does. Not all return instructions in program A produce a return outcome, some are liable to generate conditions which are detected by the Access Guardian or an inherent MC68000 detection mechanism and generate a restart outcome. The effect of the software detection mechanisms is

(a) Version 1 : Original Program

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	13	7	4	5	4	4.5	2.5	3	0	0
UJ	0	16.5	14	14	16.5	12	6.5	4.5	4	2.5	1
RN	0	2	2	3	3.5	2	1.5	1	1.5	1	0.5
SW	0	0	0	0	0	0	0	0	0	0	0

(b) Version 2 : Insertion of Default Size Detection Mechanisms

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	93	17	5	1	0	0	0	0	0	0
UJ	0	32.5	37.5	33	15	8.5	4	0.5	0	0	0
RN	0	2	2	0.5	1	0.5	0	0	0	0	0
SW	0	0	0	0	0	0	0	0	0	0	0

(c) Version 3 : Insertion of Optimum Size Detection Mechanisms

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	51	16	5	0	0	0	0	0	0	0
UJ	0	32.5	35.5	33	15	8.5	4	0.5	0	0	0
RN	0	2	2	0.5	1	0.5	0	0	0	0	0
SW	0	0	0	0	0	0	0	0	0	0	0

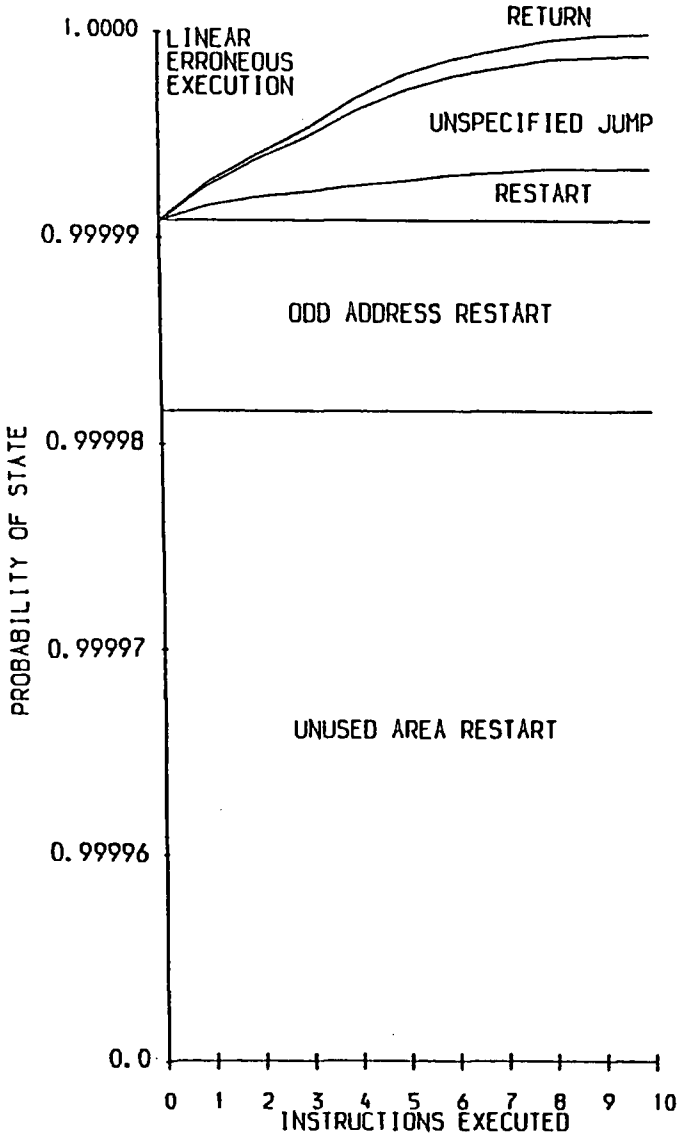
Notes:

i) RT - Restart Outcome
ii) UJ - Unspecifies Jump Outcome
iii) RN - Return Outcome
iv) SW - Stop/Wait Outcome

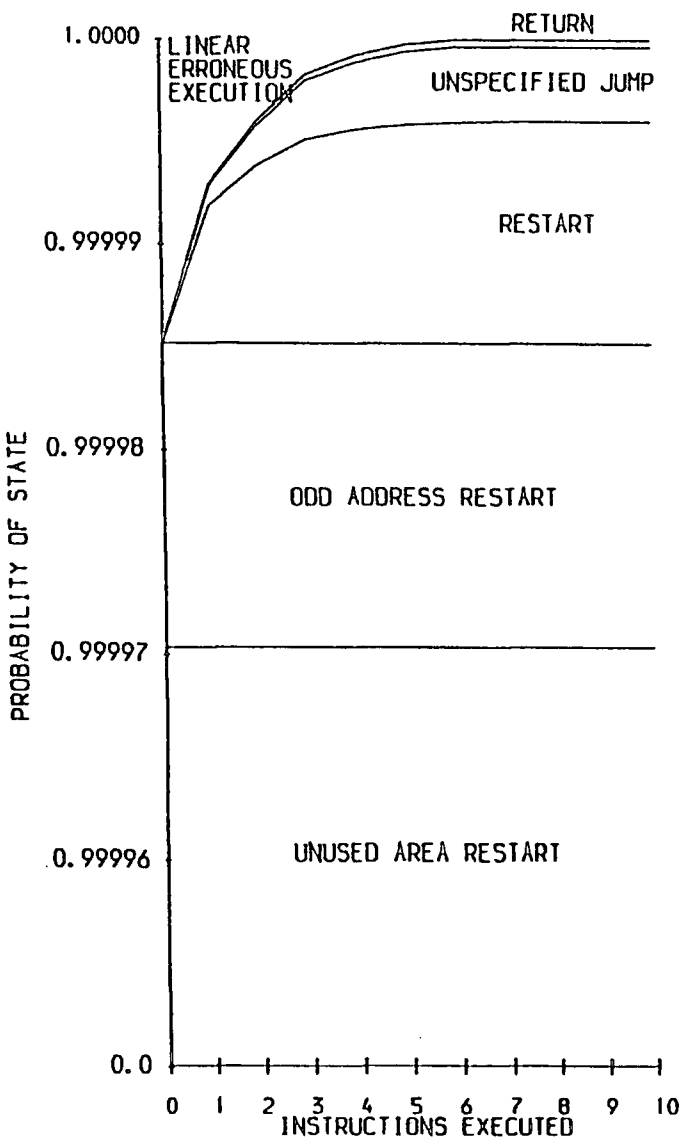
Jump Outcome Statistics

i) There are a total of 32 potential jump instructions open to interpretation (version 1).
ii) 15 valid branches are specified within the original program.
iii) None of the invalid branches can be detected by an Access Guardian.
iv) 17 invalid branches can be detected by the insertion of 16 software detection mechanisms (version 2 and version 3).

Table 7.4. : Observed Behaviour of Program 'A'



(a) Version '1'



(b) Version '2'

Figure 7.4. : Program 'A' - IEJ Execution Phase

clearly observed in version 2 in which approximately half of the unspecified jump outcomes now generate a restart outcome, the remaining unspecified jumps occurring within re-synchronization.

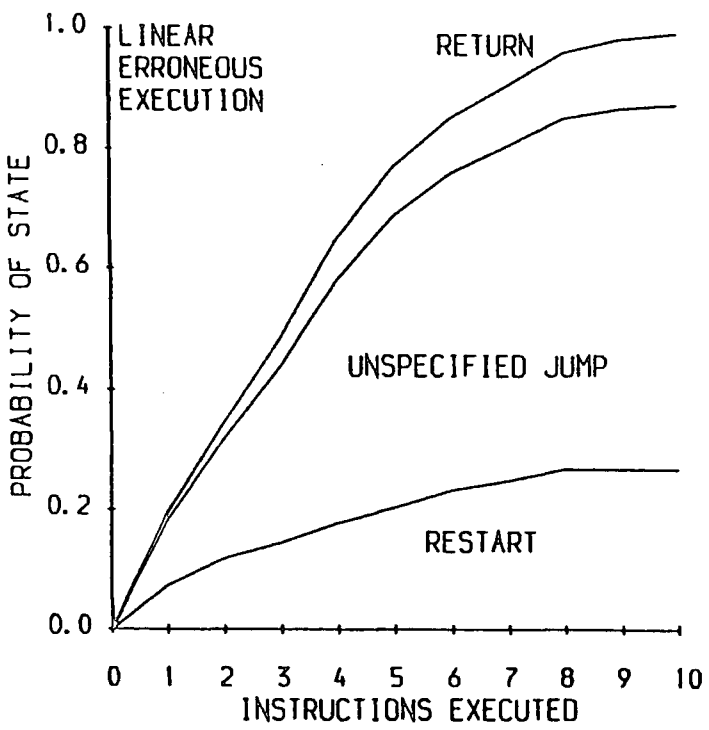
7.4.5.2. The Subsequent Erroneous Jump Phase

The observed Subsequent Erroneous Jump (SEJ) phase of erroneous execution for version 1 and 2 of program A are shown in Figure 7.5. The observations made for the SEJ phase are the same as those for the IEJ phase except that the ordinate intercept is origin. This is because the detection capabilities of the Access Guardian and the MC68000 odd byte address exception are taken into account by the detection outcome of their generating instructions in the previous phase of erroneous execution.

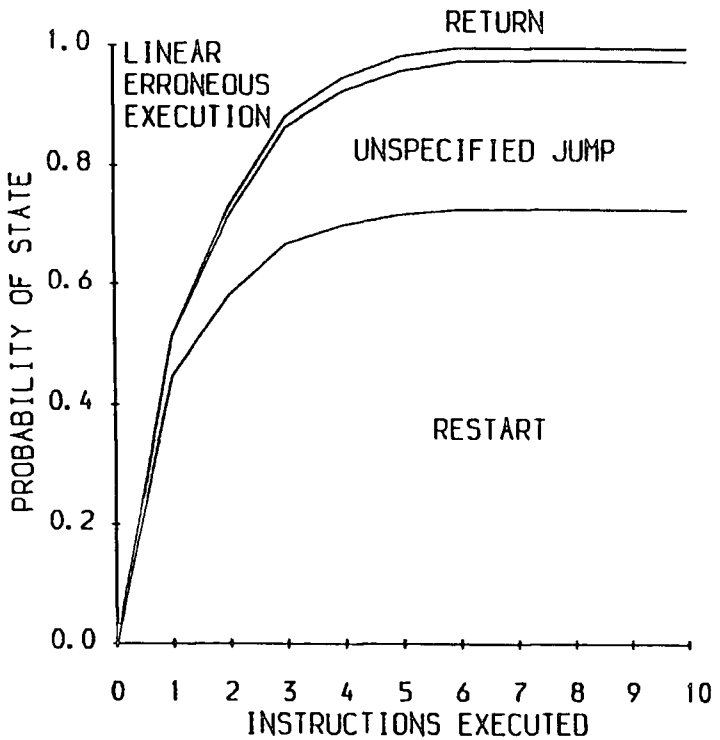
The period of linear erroneous execution following an SEJ in version 1 is typically longer than that for version 2. This is denoted in Figure 4 by the combined cumulative jump outcomes for version 1 reaching approximately 100% after 10 instructions processed, compared to 6 instructions processed for version 2. Furthermore, the percentage of SEJ phases terminated by a restart outcome, representing detection, is greater for version 2 than version 1. This observation clearly demonstrates the enhanced detection capability, provided by the insertion of software detection mechanisms, in the SEJ phase.

7.4.6. Analysing Detection Capability

The data collected for the two phases of erroneous execution are inserted into equation (3.13.) in order to determine the dynamic detection capability of the software based fault tolerant technique. The fault emulation results for program A, B, and C, shown in Tables 7.4., 7.5., and 7.6., are processed by equation (3.13.) to produce Figure 7.6(a, b, c). respectively. Each figure shows the detection capability for version 1, 2, and 3 of the program. The programs are assumed to be implemented on a microprocessor system with an embedded Access Guardian. This assumption facilitates evaluation of program counter faults covering the whole address space without the need to model and emulate the unknown data content of the unused address space.



(a) Version '1'



(b) Version '2'

Figure 7.5. : Program 'A' - SEJ Execution Phase

(a) Version 1 : Original Program

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	2.5	1.5	0.5	0	0	0	0	0	0	0
UJ	0	15	10	6.5	4.5	5	3	1.5	0.5	0.5	0
RN	0	2	0	0	0	0	0	0	0	0	0
SW	0	0	0	0	0	0	0	0	0	0	0

(b) Version 2 : Insertion of Default Size Detection Mechanisms

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	20.5	13.5	6.5	2.5	1.0	0	0	0	0	0
UJ	0	19.5	8	5.5	3	3.5	2	0.5	0	0	0
RN	0	2	0	0	0	0	0	0	0	0	0
SW	0	0	0	0	0	0	0	0	0	0	0

(c) Version 3 : Insertion of Optimum Size Detection Mechanisms

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	14.5	13.5	6.5	2.5	1.0	0	0	0	0	0
UJ	0	19.5	8	5.5	3	3.5	2	0.5	0	0	0
RN	0	2	0	0	0	0	0	0	0	0	0
SW	0	0	0	0	0	0	0	0	0	0	0

Notes:

i) RT - Restart Outcome
ii) UJ - Unspecifies Jump Outcome
iii) RN - Return Outcome
iv) SW - Stop/Wait Outcome

Jump Outcome Statistics

i) There are a total of 22 potential jump instructions open to interpretation (version 1).
ii) 7 valid branches are specified within the original program.
iii) 5 invalid branches can be detected by an Access Guardian (version 1).
iv) 9 invalid branches can be detected by the insertion of 9 software detection mechanisms, the remaining invalid branches created by the inserted software detection mechanisms can be detected by the Access Guardian (version 2 and version 3).

Table 7.5. : Observed Behaviour of Program 'B'

(a) Version 1 : Original Program

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	10	7	2	1	0	0	0	0	0	0
UJ	0	32.5	39.5	45	34	21	16.5	14.5	10	11	7
RN	0	3	3.5	2	0	0	0	0	0	0	0
SW	0	0	0	0	0	0	0	0	0	0	0

(b) Version 2 : Insertion of Default Size Detection Mechanisms

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	76	7	2	1	0	0	0	0	0	0
UJ	0	32.5	42.5	47.5	34.5	19	15.5	14.5	10	11	7
RN	0	3	3.5	2	0	0	0	0	0	0	0
SW	0	0	0	0	0	0	0	0	0	0	0

(c) Version 3 : Insertion of Optimum Size Detection Mechanisms

Jump Outcome	Number of Instructions Processed										
	0	1	2	3	4	5	6	7	8	9	10
RT	0	19	7	2	1	0	0	0	0	0	0
UJ	0	32.5	42.5	47.5	34.5	19	15.5	14.5	10	11	7
RN	0	3	3.5	2	0	0	0	0	0	0	0
SW	0	0	0	0	0	0	0	0	0	0	0

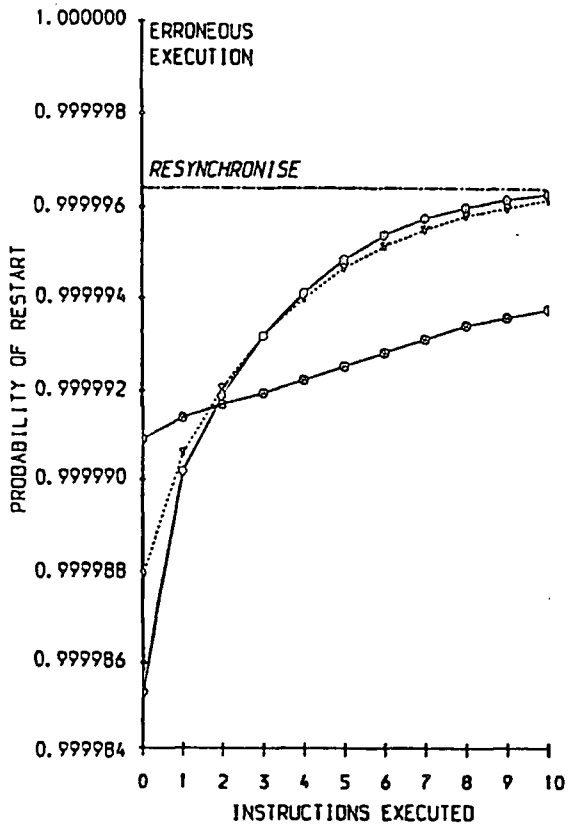
Notes:

i) RT - Restart Outcome
ii) UJ - Unspecifies Jump Outcome
iii) RN - Return Outcome
iv) SW - Stop/Wait Outcome

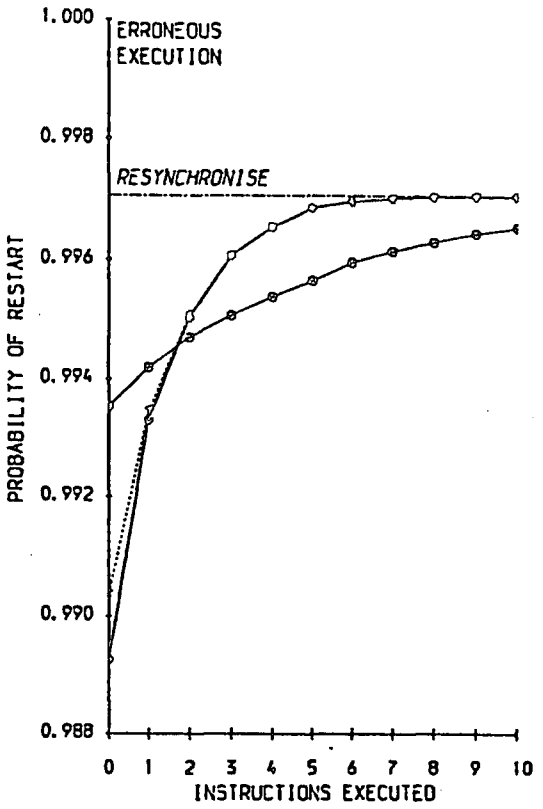
Jump Outcome Statistics

i) There are a total of 35 potential jump instructions open to interpretation (version 1).
ii) 27 valid branches are specified within the original program.
iii) None of the invalid branches can be detected by an Access Guardian.
iv) 7 invalid branches can be detected by the insertion of 6 software detection mechanisms (version 2 and version 3).
v) There is 1 non-critical placement deadlock.

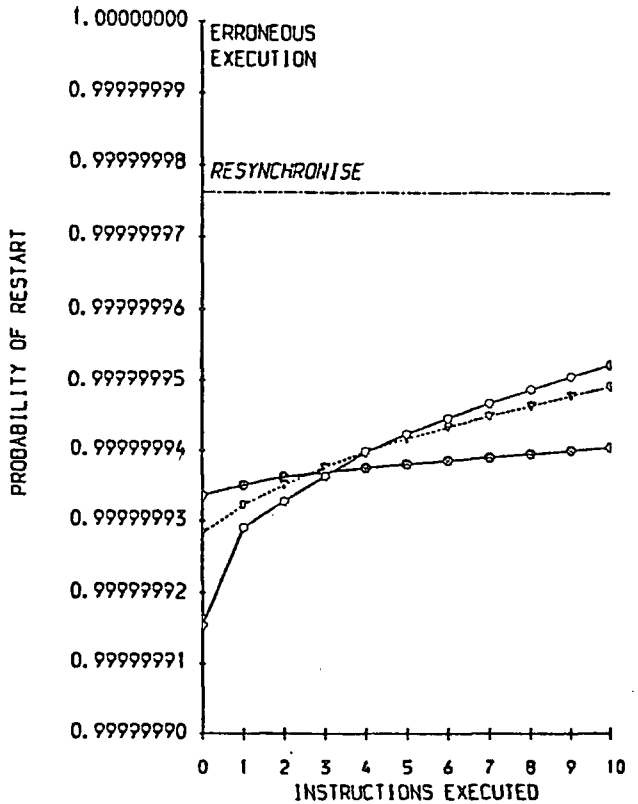
Table 7.6. : Observed Behaviour of Program 'C'



(a) Program 'A'



(b) Program 'B'



(c) Program 'C'

Figure 7.6. : Detection Capabilities of Example Programs

KEY

⊙ VERSION 1 : ORIGINAL

▽ VERSION 2 : DEFAULT DETECTION MECHANISM

○ VERSION 3 : OPTIMUM DETECTION MECHANISM

The relationship between the three versions of each of the programs is very similar. There is a detection capability enhancement shown by versions 2 and 3 over version 1, and version 3 over version 2. However, there is an initial performance overhead associated with inserting software detection mechanisms. This is clearly seen in Figures 7.6(a, b, c). where for the first one or two erroneously processed instructions the detection capability of versions 2 and 3 is lower than that for version 1 (the original) of the program. This performance overhead is due to the increased program size and hence the higher probability that the corruption of the program counter, initiating the IEJ phase of erroneous execution, takes a value corresponding to a location within the program. The performance overhead is quickly over-ridden by the enhanced detection capability provided by the software detection mechanisms.

The relative overhead of the version 2 compared with version 1 depends on the number and default size of the detection mechanisms. The number of detection mechanisms that can be placed is dependent on the code content of the application program. The default size of detection mechanisms is dependent on the application processor's instruction set, and the optimum size required for each placement is dependent on the local code structure. The overhead associated with version 3 is dependent of the memory saving facilitated by using the optimum size of detection mechanism on each placement. There is little difference in versions 2 and 3 for program B because the mean optimum detection mechanism size is about the same as the default detection mechanism size. There is a larger difference in program A, and larger again in program C, because the mean optimum detection mechanism size is smaller than the default detection mechanism size.

The effectiveness of the software detection mechanisms is dependent on their number and distribution within the application program. Program B yields some of the best results due to a large number of software detection mechanisms spread evenly throughout the code. Program A obtains similar benefits from the insertion of software detection mechanisms. Program C results are poorer because fewer software detection mechanisms could be placed, and there is a contiguous block the size of

half the application program which is void of mechanism placements. The effect on program C is to greatly slow down the detection capability.

It will also be noticed that the version 3 results, whilst having a smaller overhead have a slightly reduced detection capability, compared with version 2, after several instructions have been erroneously processed. This is clearly seen for programs A and C in Figure 7.5. The variation in detection capabilities is due to erroneous execution flow, other than a SEJ, being detected by the software detection mechanisms. The default size detection mechanisms detect all non-re-synchronized linear erroneous execution because they have the same number of seed bytes as bytes in the longest instruction construct for the application processor. Hence, it is guaranteed that during linear erroneous execution a seed will be interpreted as an opcode, generating a restart outcome and detection. Optimum size detection mechanism will not detect all linear erroneous execution passing through the detection mechanism because some erroneously interpreted opcodes will consider all the seeds as operands of the current instruction.

Re-synchronization of erroneous execution within the experiment generates a detection capability ceiling. This ceiling is rapidly approached by programs A and B, due to the large number of inserted software detection mechanisms. Program C did not facilitate the same potential for software detection mechanism insertion because of the limited number of invalid branches, and hence the detection capability enhancement is not so rapid. Nevertheless, all three programs show improved detection capability. For highly reliable systems, additional techniques should be employed to cover the probability of re-synchronized erroneous execution.

7.4.7. Critical Hazards of Erroneous Behaviour

Three main classes of critical hazard are identified: cessation of processing (described in Chapter 3), placement deadlock (described in Chapter 5), and infinite execution loops. The observations and implications of these hazards within the example programs A, B, and C are now discussed. The next chapter in this thesis considers a technique for removing these hazards.

7.4.7.1. Cessation of Processing

None of the example programs included code which could be interpreted during erroneous execution as a stop/wait outcome and hence cause microprocessor operation to cease. Program B does, however, include instruction sequences which require user input. These instruction sequences could generate an apparent cessation of processing if erroneous execution re-synchronizes at one of these instruction sequences - the user not being aware of the required input. Such occurrences are very difficult to prevent and an additional fault tolerant methodology is required to remove the hazard.

7.4.7.2. Infinite Execution Loops

These hazards involve the creation of an infinite loop by erroneous jumps. Once erroneous execution enters such a structure, the correct function of the program is permanently lost. The hazard is removed by implementing the software based fault tolerant technique proposed in the thesis, involving the placement of software detection mechanisms and the attachment of an Access Guardian to the microprocessor system as required. A good example of this hazard is demonstrated in program C. The assembler listing of the program can be found in Appendix D. An infinite loop is generated by the erroneous jump at location $1D_{hex}$ in the 'getvalue' routine, as shown in Table 7.7.

7.4.7.3. Placement Deadlock

The potential hazard of a placement deadlock occurred once during the application of software implemented fault tolerance documented in Appendix D. Placement deadlock describes the situation where an erroneous jump has its generator and destination are operands within the same instruction. The identified placement deadlock, shown in Table 7.8., is at location $2E_{hex}$ within the 'main' routine in version 1 of program C (listed in Appendix D). Fortunately this potential erroneous jump is not backward, and hence there is no danger of an erroneous infinite execution loop.

Placement deadlocks can, however, be hazardous and it is pertinent to develop techniques for manipulating code with this attribute.

Intended Execution	Program Segment		Erroneous Execution
	Address	Code	
addl $8,%esp	001B	83	
operand	001C	C4	
operand	001D	08	OR *and* ← JNE *at* 001F
movl -4(%ebp),%eax	001E	FF	*operand*
operand	001F	75	JNE → 001D
operand	0020	FC	*operand*

Table 7.7. Erroneous Infinite Execution Loop

Intended Execution	Program Segment		Erroneous Execution
	Address	Code	
call swap	002D	E8	
operand	002E	7A	JP → 002F
operand	002F	FF	operand *and* ← JP *at* 002E
operand	002F	FF	
operand	002F	FF	
operand	002F	FF	

Table 7.8. Placement Deadlock

7.4.8. Re-synchronizing Erroneous Execution

The re-synchronization experienced by version 1 of the example programs is influenced by the relative number of opcodes to operands. Lower ratios encourage re-synchronization, because there are more opcodes available for interpretation as such in the range of locations open to instruction translation during erroneous execution. Version 1 of programs C, A, and B have ascending speeds of re-synchronization, with descending opcode/operand ratios respectively. The opcode/operand ratio is dependent on the instruction mix within the application program, and the instruction constructions within an applications processor's instruction set.

7.5. Summary and Conclusions

This chapter documents the results of two experimental programmes. The first involved injecting 2136 faults into a MC68000 microprocessor-based system and monitoring the system's response. The second involved emulating the response of a MC68(7)05, a MC68000, and an Intel 80386 microprocessor-based system with a combined total of 1812 emulated faults.

The faults selected for the experiments were single and multiple bit. Several microprocessor systems with various detection facilities are monitored in order to evaluate their response to each fault insertion. The behavioural observations highlight the re-synchronization phenomenon whereby erroneous execution, diverted by the activation of an inserted fault, returns to a valid program path. In particular two types of hazard are associated with re-synchronization: placement deadlock, and sleeping corruption of the processor stack. Access to a corrupted stack re-initiates erroneous execution and detection is facilitated by the fault tolerant mechanisms resident in the experimental system. Cessation of processing and placement deadlock are identified as not being covered by the implemented fault tolerant mechanisms, and require other techniques to remove their hazards. Such techniques are proposed and discussed in the next chapter.

Fault injection experiments and fault emulations have indicated the effectiveness of the software based fault tolerant technique proposed in Chapter 5. Performance parameters used to assess the technique are largely dependent on the nature of the

application program to be implemented with the microprocessor system under evaluation. The only overhead which can be accurately stated is the hardware overhead attributed to the design of the Access Guardian, and this is discussed in Chapter 5. Therefore although figures for error coverage, detection latency, and processing overhead are derived for application programs, their replication for other program implementations is not assured. This is a point not sufficiently stressed by other reports of fault injection, simulation, or emulation experiments.

Nevertheless, the benefits of implementing the software based fault tolerant technique have been clearly demonstrated for those microprocessor systems under evaluation. In addition, the behaviour of a microprocessor's erroneous execution has been observed. These observations have shown the importance of re-synchronizing erroneous execution, and have validated the erroneous microprocessor behaviour model proposed in Chapter 3.

Chapter 8

GENERATING NON-HAZARDOUS SOFTWARE

CHAPTER EIGHT

GENERATING NON-HAZARDOUS SOFTWARE

8.1. Introduction

Previous chapters have identified the characteristics of microprocessor behaviour following an event disrupting software execution. A fault tolerant technique has been developed to detect erroneous behaviour and initiate recovery. There still, however, remains the uncertainty of the nature of any erroneous behaviour.

Erroneous microprocessor execution is dependent upon the operation outcomes of interpreted instructions through the address space. There may be associated risks with erroneous execution which adversely effects system integrity. In particular, erroneous execution may process critical hazards which are indicative of catastrophic processing failures.

Translators can have considerable influence on the nature of target machine code generated to implement a high-level language program [Ciminiera & Valenzano, 1987]. Machine code is produced primarily on the criteria of performance and efficiency, however, such code may incorporate critical hazards as observed in the previous chapter. Microprocessor reliability can be considerably improved if hazardous code is not generated.

This chapter investigates the generation of software by two types of translator: compilers and assemblers. Generated software consists of machine code for program, data, and input/ output reserved areas of the address space. Techniques are proposed for each of these areas to manipulate the machine code in order to prevent the generation of critical hazards within the code released by a translator. To facilitate these techniques, it is necessary to encourage and discourage particular translator practices. These practices are influenced by the characteristics of the target microprocessor. A selection of 8, 16, and 32-bit microprocessors are examined including the Intel 8086 and Motorola 68000 families, and their implications for translator action discussed.

8.2. Bridging the Semantic Gap

Microprocessor design is advancing so rapidly that design-oriented software can quickly become obsolete. It is therefore important that software should be written in a manner that facilitates implementation on a variety of processors. Even with upwardly compatible microprocessor designs, implementation of an original piece of software inefficiently uses the capability of the advanced design.

High-level languages have been developed which are abstracted from processor architectures and configurations to the extent that the language is independent of the target machine. Examples of high-level languages include FORTRAN, Pascal, and C. They are considered portable languages because they allow software to be implemented on a range of processor types.

The difference between a high-level language and its target microprocessor implementation is known as the *semantic gap*. Bridging this gap has a major influence on the overall execution efficiency and software reliability of the microprocessor application. The tool used to bridge the gap is the translator. Translators automatically process the high-level software down through the levels of abstraction until code is generated which is executable on the target processor. The nature of translation means that the programmer has no control over the generation of the machine code and any hazards it may contain. It is important that translators are designed to generate non-hazardous machine code.

8.3. The Risks of Erroneous Execution

8.3.1. Catastrophic Processing Failures

Two particular hazards identified with catastrophic failure are the possibilities of cessation of processing, and infinite processing loops during erroneous execution. Both occurrences prevent the possible detection of erroneous microprocessor behaviour by software techniques: an external hardware reset being necessary to restore the microprocessor system.

Processing ceases when an instruction execution outcome is a stop/wait state. Exit from this state requires external intervention. The probability of process cessation can be predicted by examining individual address space elements for stop/wait instruction codes. However, it must be recognized that the processing action of the processor with volatile memory can introduce further occurrences of this hazard which cannot be predicted.

Infinite execution loops are instruction sequences that are continuously executed in a cyclic fashion. There is no processing exit from these loops except through some external intervention. Such loops may incorporate a chain of erroneous jumps, and operate through a combination of different functional areas in the address space (described in Chapter 5). Identification of all infinite execution loops for application software would involve tracing the execution path for all possible erroneous execution variants following an Initial Erroneous Jump (IEJ). In reality, execution paths may change. A loop might include a conditional branch instruction. Erroneous execution might change the tested condition codes and hence the loop would not be infinite. Prediction of infinite execution loops would also have to allow for a section of address space being implemented on volatile or non-volatile memory. Obviously volatile memory is subject to manipulation through processor activity such as stack operations.

8.3.2. Critical Hazard Coverage

In order to prevent the occurrence of catastrophic processing failures, it is necessary to provide coverage for all associated critical hazards. The two identified hazards of catastrophic failure, cessation of processing and infinite execution loops, rely on the action of erroneous jumps. It has been shown in the previous chapter that translator generated software can include hazardous code which if interpreted as an opcode produces an erroneous jump. The generation and eventuality of executing such hazardous code is covered to improve the reliability of the microprocessor system.

A fault tolerant technique based on software enhancement has been proposed in Chapter 5. This technique provides rapid termination of erroneous behaviour by

detecting the execution of erroneous jumps. The technique implements an Access Guardian which provides immediate detection of erroneous execution in the unused area of the address space. The effectiveness of the technique in the used area depends on the number of erroneous jumps within the machine code. However there are three particular erroneous jump constructs which cannot be covered by the the fault tolerant technique : stop/wait erroneous jump outcomes, return erroneous jump outcomes, and placement deadlock.

The interval between the initiation of erroneous execution and its detection is called error latency. During this period there is a risk of a critical hazard not covered by the fault tolerant technique being executed. Such processing may generate a catastrophic failure.

Clearly there is a need to develop a translator which avoids generating these classes of erroneous jump hazard. Such a translator would reduce the inherent hazard associated with the code without reducing the detection improvement offered by application of the fault tolerant technique. Indeed the ability to manipulate used area code enables not just the prevention of hazards, but also the introduction of attributes facilitating detection of erroneous execution.

8.4. Non-Hazardous Program Area Code

8.4.1. Hazardous Instruction Formats

The program area consists of instruction sequences generated by the translator. Each instruction has a conceptual structure consisting of a descriptor and an address field. The descriptor specifies the instruction operation. The address field specifies the information to be processed by the instruction. The manner in which the address field is interpreted is known as the addressing mode of the instruction. The content of the address field is referred to as the stated address. The address of the referenced memory location, containing the information to be processed, is referred to as the effective address.

The physical implementation of an instruction consists of opcodes representing the descriptor, and operand(s) if required by the addressing mode representing the

stated address. Opcodes are defined in the microprocessor's instruction set. Operands are specified by the addressing modes implemented within the microprocessors architecture. The most common addressing modes are

Inherent Addressing :	No effective address required.
Direct Addressing :	The stated address is the effective address of a register.
Immediate Addressing :	The effective address is the location immediately following the opcode.
Indexed Addressing :	The effective address is the stated address added to the contents of a specific register (*index register*).
Indirect Addressing :	The stated address serves as a pointer to the location at which the effective address resides.
Register-Indirect Addressing :	The stated address serves as a pointer to a specified register which contains the effective address.
Relative Addressing :	The stated address is an offset which is added to the contents of a location inherently selected by the opcode to form the effective address.

The hazard identified with catastrophic processing failures, the erroneous jump, can be generated by opcodes and operands. The addressing mode adopted by an instruction influences the requirement and nature of operands. Techniques are proposed for the selection of instructions and their manipulation to prevent the generation of critical hazards. To this end, certain translator practices can be encouraged or discouraged. Several practices are identified and discussed.

8.4.2. Hazardous Opcodes

Early microprocessor designs such as the Intel 8080 and Motorola 6800 implement an instruction set in which all operations can be uniquely identified by a single-byte opcode. Hence these opcodes have no associated hazard.

Later processor designs extended the instruction set to improve flexibility and performance. Examples of this type of machine include the Motorola 68000 and Intel 8086. Such microprocessors are known as Complex Instruction Set Computers (CISC). Processor designs that extend the instruction set beyond 256 unique operations require a multiple-byte opcode. However, the second and subsequent opcode bytes are liable to have a hazardous format susceptible to erroneous execution. The nature of this hazard can be determined by mapping the opcode bytes onto the first-byte opco map.

The Intel 8086 microprocessor family have instructions which require an additional opcode byte to further specify the operation code of the first opcode byte. Hazardous bytes can be identified and suitable translator action taken.

The Intel 80286 microprocessor instruction set has an instruction 'SETNP' whose second byte, if interpreted as an opcode, generates a stop/ wait outcome. This is a critical hazard associated with catastrophic processing failure. There is no method of manipulating the second-byte of the opcode because its format is completely defined. The selection of this and other instructions that have hazardous opcode bytes that cannot be manipulated should be avoided in order to prevent the possibility of catastrophic failure during erroneous execution.

All instructions in the Motorola 68000 microprocessor family instruction sets have a two-byte opcode. Furthermore, these microprocessors implement memory organization which specifies that instructions reside at even-byte boundary addresses. Interpretation of an instruction at an odd-byte boundary address generates a software exception and hence a restart outcome. Hence there is no hazard associated with the second opcode byte because of the over-ride action of the odd-address exception.

Some modern microprocessor designs have returned to a smaller instruction set, discarding under-utilized instructions. The core of the instruction set facilitates efficient processing within a simplified machine architecture. The processors are known as Reduced Instruction Set Computers (RISC). An example of this processor type is the AMD Am29000. This microprocessor specifies its instructions to have a single-byte opcode and a three-byte operand, with the notable exception of the load and

store instructions where there is a two-byte opcode and a two-byte operand. Single-byte opcodes have no hazard whilst two-byte opcodes may have a hazardous second byte.

8.4.3. Hazardous Operands

8.4.3.1. Prevention of Addressing Mode Hazards

In general, instructions have operands which are susceptible to interpretation as a first-byte opcode during erroneous execution. The addressing mode adopted by an instruction influences the requirement and nature of these operands. Instructions may be selected on the basis of the non-hazardous operands associated with addressing mode and their ease of manipulation.

An exception to the rule is the AMD Am29000 microprocessor. This machine locates instructions at every other even-byte address. A exception facility can be enabled within this processor such that a program counter, containing the address of an operand byte, is masked to access the opcode address. Hence the operands may have a hazardous format but the microprocessor architecture prevents their interpretation during erroneous execution. No operands, therefore, require manipulation to remove any hazard.

The Intel 8086 microprocessor family architecture implements segmentation. Every memory access requires the specification of a segment register in conjunction with an addressing mode. The segment register and the addressing mode implement the most and least significant address bits respectively. Hence all addressing modes facilitated by these processors are pseudo-relative. The allocation of segment registers can be automatic which eases the complexity for low-level programmers and translators alike.

8.4.3.2. Inherent Addressing

In this addressing mode there is no effective address requirement. Inherent addressing instructions have all necessary information for processing within the instruc-

tion opcode. No operands are needed. A common inherent addressing instruction to many microprocessor instruction sets is 'no-operation'.

8.4.3.3. Manipulating Direct Addressing

The number of operands required by an instruction implementing direct addressing to specify the effective address is dependent upon the size of the microprocessor address space and the size of each operand. For instance, a 4 GByte address space requires 4 bytes to specify an absolute address, whilst a 64 KByte address space requires only 2 bytes. Operands typically take a one-byte structure. However, larger byte structures are implemented by particular microprocessors, notably the Motorola 68000 family which have two-byte operands. Each operand used to represent the effective address may have a hazardous format.

The hazard associated with an operand that partially or fully represents an effective address is dependent on that address. Accessed memory can be relocated so that the operands specifying the effective address take non-hazardous code formats.

To give the operands of an instruction implementing direct addressing a restart format, it is necessary to have groups of consecutive restart opcodes within the microprocessor instruction set. Larger groups of restart opcodes reduce the complexity of movement of the used area when covering each directly addressed location. The Intel 8086 microprocessor like the AMD Am29000 has an instruction set which has few useful restart instruction formats. The advanced Intel 80286 and 80386 microprocessors, which are upwardly compatible with the 8086, have no additional restart instructions of a useful format. The Motorola 68000 microprocessor family have a large number of useful restart instruction formats. Their instruction sets of 65536 instructions comprise of approximately 2% defined restart instructions, and 30% undefined instructions specified to generate a software exception which is a restart outcome. Within these restart formats are two large blocks taking the hexadecimal values AXXX and FXXX, where 'X' is a 'don't care' hexadecimal value. These represent undefined instructions which are reserved for implementation by later releases of the microprocessor family. In particular, the Motorola 68030 microprocessor facilitates the use of some FXXX hexadecimal format instructions for co-processor operation.

However, if the microprocessor system does not incorporate a co-processor then these instructions revert to generating a restart outcome. Each of the two blocks of restart instructions when converted to an address range cover 4 KBytes. The remainder of the instructions generating a restart outcome are scattered about the opcode map.

8.4.3.4. Manipulating Immediate Addressing

In the immediate addressing mode, the effective address is, by default, the instruction operand(s). The operands contain data to be processed by the instruction.

The operands are by their nature application specific. Their hazards are dependent on the host microprocessor instruction set. Although the data stored cannot be altered, the format of its storage can be manipulated.

A technique is proposed for microprocessors whose architecture organizes memory on a double byte addressing scheme such as the Motorola 68000 family. Operands therefore have a two-byte format. For single byte data, one byte holds the data, while the remaining byte can be set to a value such that the resultant operand code is non-hazardous. The choice of operand values that do not exhibit a hazard is dependent on the target instruction set.

8.4.3.5. Manipulating Indexed Addressing

This mode of addressing requires operands to represent the stated address and the index register. This addressing mode is particularly useful when implementing a stack structure in memory for microprocessor operation.

Some microprocessor architectures, including the Motorola 68000 family, specify register use within the instruction opcode. Other architectures, such as the Intel 8086 family require an operand to specify register usage. In these instances it is suggested that some registers may be preferred. The generation of operand code specifying particular registers and which has a hazardous format should be avoided.

The operand(s) representing the stated address can be manipulated using the same techniques proposed for direct addressing: operands specifying absolute addresses for direct addressing should not have a hazardous bit format.

The Intel 8086 and Motorola 68000 microprocessor families implement additional variants of the indexed addressing mode. These involve a base address displacement which is added to the effective address. Such extensions to the indexed addressing mode facilitate enhanced data processing techniques.

8.4.3.6. Manipulating Indirect Addressing

The stated address contained in the operands attached to an instruction implementing indirect addressing is an intermediate location in the address space. The content of this location specifies the effective address of the data to be processed by the instruction.

It is proposed that an intermediate location is selected so that the operands specifying its address do not take a hazardous value. However, the intermediate locations available may be restricted by the residence of used areas in the address space. Movement of the used area can release useful locations.

It is valuable when manipulating indirect addressing, to have a large number of scattered opcode formats for restart instructions. This reduces the likelihood of having to move sections of the used area in order to release addresses that mimic a restart opcode format.

Both the Intel 8086 and Motorola 68000 microprocessor families do not implement indirect addressing, preferring to use the equivalent register-indirect addressing mode.

8.4.3.7. Manipulating Register-Indirect Addressing

This method of addressing is closely allied to the indirect addressing mode. The effective address is contained within a register specified by the stated address.

As identified with the index addressing mode, some microprocessor architectures require the specification of a register within an operand. These operands can be manipulated to remove any hazard *loc cit.*

This addressing mode is implemented with particular effect in respect to hazard generation in the Motorola 68000 microprocessor family. These machines implement an instruction using register-indirect addressing without an operand requirement, i.e. the opcode specifies the register. Opcodes for these processors do not have a

hazard, so instructions implementing this addressing mode do not introduce hazards and should be encouraged by translator code generation.

8.4.3.8. Manipulating Relative Addressing

The inherent register specified by relative addressing serves as a pointer to a memory location. The stated address acts as a displacement to the pointer. The result is a method of referencing memory in the vicinity of the address contained in the register.

An important class of instruction that implement relative addressing are branches. These facilitate changes in the otherwise sequential control flow of program execution. Relative addressing is used in conjunction with the microprocessor's dedicated register, the program counter, which serves as a pointer to the next instruction to be processed.

The displacement required by this addressing mode is stored in one or more operands. A large proportion of relative addressing displacements are local. Operands specifying local forward and backward displacements have low and high hexadecimal values respectively. The hazards associated with such operand values are dependent on the host microprocessor instruction set. Displacements represented by operand code of a hazardous format should not be used. Operand manipulation is only possible where the displacement value is independent of run-time conditions.

Assuming that branch offsets tend to be local, deductions can be made for different microprocessor implementations. This is known as the principle of program locality [Ciminiera & Valenzano, 1987]. Two popular microprocessor families are investigated with respect to local relative addressing hazards.

Firstly, consider a Motorola 68000 target machine. Local backward branches will produce an operand of the hexadecimal format FFXX, an undefined instruction, which, when executed as an opcode, generates a restart outcome and hence no hazard. Local forward branches produce an operand of the format 00XX which, when executed as an opcode, is highly likely to produce an OR instruction. Although there is no immediate hazard associated with an OR instruction it does continue erroneous execution, and may corrupt system data. Implementation on the compatible but

extended instruction set of the Motorola 68020 realizes usage of some of the FFXX instructions for co-processor operation. When a co-processor is incorporated into the microprocessor system, local backward jumps have a probability of producing an erroneous instruction execution hazard. However, if a co-processor is not present then these instructions continue to generate a restart outcome.

Secondly consider the Intel 8086 microprocessor family. Local backward branches tend to produce FF hexadecimal values for the most-significant, and high hexadecimal values for the least-significant operand bytes. Equally, local forward branches tend to produce 00 hexadecimal values for the most-significant, and low hexadecimal values for the least-significant operand bytes. The byte FF hexadecimal value, when erroneously interpreted as a first-byte opcode specifies a multi-byte opcode, which can take a format associated with critical hazards. The byte 00 hexadecimal value, when mapped onto the first-byte opcode map, reveals that erroneous execution will generate an ADD instruction. Whilst this is not classified as a critical hazard, it still allows erroneous execution to propagate and perhaps activate a critical hazard elsewhere. Low and high hexadecimal values can thus generate both detection of erroneous execution and catastrophic processing failures. Hence local branches are fraught with danger for translators generating machine code for the Intel 8086 microprocessor family. Translators can move sections of code within the address space to ensure bytes specifying local branches do not take formats defining critical hazards. The intermediate vacant slots of memory between relocated sections of code can be filled with 'no-operation' single-byte instructions, or sections of code linked by a branch instruction and intermediate locations filled with restart single-byte restart outcome instructions yielding a detection capability. This proposed manipulation of the program area will introduce a processing performance overhead to the correctly executing program. However, this overhead is deemed to be acceptable for most microprocessor applications.

8.5. Influencing Translator Practices

8.5.1. Instruction Selection

It is desirable that a translator uses a microprocessor instruction set efficiently during its code generation phase. That is, program code is produced for optimal performance. Ineffective use of the instruction set could lead to unnecessary increases in the processing time and memory space required by the program code.

Selection of instructions which process registers has a particular benefit. Addressing modes using registers do not require a memory fetch and hence are more efficient than addressing modes using direct addressing. Instructions which process registers thus enhance program code performance.

An example of translator inefficiency is demonstrated by the UNIX 'cc' compiler. The target machine is the Motorola 68000. The instruction set for this microprocessor includes a branch instruction with an embedded byte displacement, for relative addressing, in the opcode. This compiler, however, generates a branch instruction with a two-byte displacement for a branch requiring a single byte displacement. The implemented instruction requires an otherwise unnecessary operand. Resultant program code requires extra memory, and has a slower execution for this instruction.

Whilst generated code should be efficient, it should also not exhibit any critical hazards. Manipulation of the code to remove these hazards may degrade program performance. Particular translator actions can be encouraged or discouraged so that the generated code is acceptable.

8.5.2. Coupling and Cohesion

Selected instructions should exhibit the characteristics of coupling and cohesion. These characteristics are more commonly associated with high-level language software engineering [Sommerville, 1985]. Coupling is a measure of the programs code dependence on referenced parameters. Cohesion is a measure of the functionality (unity of operation) of the program code. Translators should generate program code that exhibits both high coupling and cohesion characteristics.

Low coupling indicates the use of specified parameters rather than referenced parameters. Specified parameters give operand formats which may have an associated hazard. Program code may replicate use of the parameter and hence the hazard is propagated. An example of this is demonstrated by the UNIX 'cc' translator on the SUN machines whose target processor is the Motorola 68020. The translator generates instructions with an immediate addressing mode to access repeated data values, and hence any hazard associated the data value is replicated. Translators can avoid this problem by implementing reference parameters. Reference parameters are independent of the specified parameter value. The reference can be altered so that its operand format does not have an associated hazard. Such manipulation does not affect the specified value of the referenced parameter. High coupling is, therefore, a useful attribute when manipulating the program code to remove hazards.

Cohesion reflects the efficiency of the generated program code. Low cohesion indicates unnecessary control flow in the generated instruction sequences. Loop invariants are a source of inefficient control flow [Aho & Ullman, 1977]. They involve a computation that produces the same result at each cycle of a loop. The computation can be moved to a point just before the loop is entered. In general, loops are a major source of program inefficiency. High cohesion is an attribute of effective program code generation.

8.5.3. Macros

A macro is a collection of target machine instructions which perform some operation not directly facilitated by the microprocessor instruction set. Macros appear in the intermediate code generated by a translator. The translator replaces the macro with its defined target instruction sequence when producing target machine code. A macro should not be confused with a procedure. Macros are called and expanded by translator action whereas procedures are called and processed by program execution.

The specification of macros should ensure no resident hazards. Removal of hazards can be achieved using the code manipulation techniques outlined in this chapter. The importance of macros increases with processor architectures implementing

smaller instruction sets which provide fewer directly executable operations. This is particularly so in RISC processors such as the AMD Am29000.

8.5.4. Peephole Optimization

Translator generated code usually goes through the process of peephole optimization before release. This is designed to enhance the performance of the program code by increasing the efficiency of small sections of code. The JPI TopSpeed (version 1) Modula-2 translator produces code for the Intel 80296 microprocessor which is particularly effective in this respect. However, optimization may involve manipulation of instruction sequences such that hazards are re-introduced. Nevertheless, peephole optimization is very valuable in generating efficient program code. Therefore, this process cannot be abandoned, but rather it must endeavour not to create any new hazards.

8.6. Non-Hazardous Data Area

These areas can include both volatile and non-volatile memory. Non-volatile memory is used to implement static data structures such as 'look-up' tables. Volatile memory can implement dynamic structures such as stacks, as well as static data structures.

Halse [1984] suggests many variants of a software seeding strategy to implement detection of erroneous behaviour in this area. Experiments demonstrate the effectiveness of seeding. However, erroneous execution in this area can have unpredictable behaviour before detection, depending on the data content. In some instances the content may change through the operation of the microprocessor system.

The AMD Am29000 microprocessor has a separate instruction and data channels, therefore instruction fetches from the data area are impossible. Interpretation of a data area as a program area generates an exception and hence a restart outcome which denotes detection of erroneous behaviour: the data area within this processor is therefore intrinsically non-hazardous.

8.7. Non-Hazardous Input/Output Reserved Area

These memory mapped areas contain locations reserved by the microprocessor architecture for communication with external devices. Used locations will contain time dependent values related to the requests to, and responses from external devices. Unused locations with an attached communication link can have an undefined content.

All memory mapped I/O locations are susceptible to erroneous interpretation as an opcode. Such erroneous behaviour may be hazardous. Manipulating the content of the I/O reserved locations can remove any associated hazard. The proposed alterations of the microprocessor system require knowledge of the I/O communication of the application software.

The values resident at used locations can be manipulated using the same technique proposed for immediate addressing. Alternatively, values sent and received from any external device could be defined to take non-hazardous formats. The communication links for the unused locations can be tied high or low, corresponding to logic 1 and logic 0 respectively. In this manner, unused memory mapped I/O addresses can be set to hold non-hazardous values.

8.8. Influence of the Instruction Set

Manipulation techniques for machine code in the used area of the address space have been proposed, based on microprocessor architecture. Specific implementation is dependent upon the instruction set of the host microprocessor. This section identifies instruction set characteristics, including content and format, which facilitate or hinder the manipulation techniques.

8.8.1. Undefined Instructions

Some microprocessors do not declare the operation of undefined instructions within their instruction set. The physical implementation of such instructions depends on the manufacturer. A manufacturer will, typically, introduce useful and varied operations for these instructions. However, there are no standard operations

for these instructions and their function can vary between manufacturers. Furthermore, the manufacturer is under no obligation to keep a particular function for different fabrications of the same microprocessor. Modern microprocessors have generally avoided this ambiguity by declaring their undefined instructions to generate a software exception - a restart outcome. Undefined instructions have therefore been given a detection attribute. Translators should encourage the use of these codes so that the detection capability of the generated code is enhanced. Modern microprocessors generally declare the operation of all possible instruction opcodes even though the use of some is undefined.

8.8.2. Restart Instructions

Instructions with a restart outcome are used to detect erroneous behaviour. Where possible code should be manipulated to attain this detection capability. The ability of a translator to endow code with this attribute is influenced by the numbers of restart instructions and their position within the target processor's instruction set.

The proportion of instructions in a microprocessor instruction set that generate a restart outcome are typically small, see Appendix A. However, in some microprocessors the undefined instructions are specified to generate a restart outcome. Undefined instructions also usually occur as groups within the instruction set, being reserved for future instruction set extensions. There can be substantial numbers of undefined instructions within an instruction set.

8.8.3. Stop/ Wait and Return Instructions

Hazards associated with code which reflect the format of a stop/wait instruction are critical. They are synonymous with cessation of processing: a catastrophic failure. Similarly, code with a return instruction format is deemed a critical hazard because it can create an infinite execution loop during erroneous execution. Such erroneous processing is also associated with catastrophic failure.

Fewer stop/wait and return instructions in the instruction set discourage the generation of critical hazards. This assumes that the translation of target machine code can be broadly considered as a pseudo-random process.

Additionally, the principle of locality, associated with relative addressing, identifies that high and low value operands occur more regularly than other formats. To further reduce the probability of hazard generation, it will be an advantage if the stop/wait and return instructions take mid-range operand values.

The generation of hazards will also be influenced by the application code access to input/output reserved areas. Specific, utilized locations in this area may have an address which has a hazardous format corresponding to a stop/wait or return instruction. The translator generation of these hazards is application specific. For this reason, microprocessors are preferred whose stop/wait and return instructions do not reflect any reserved input/output locations.

8.8.4. Unspecified Jump Instructions

Code which has a format of an unspecified jump instruction is considered hazardous in the same manner as code with a return instruction format, the hazard being indicative of an infinite execution loop and hence catastrophic failure. Although such code has an embedded critical hazard, it is a feature of asset rather than liability. The fault tolerance technique proposed in this thesis uses the occurrence of code with unspecified jump instructions formats to detect erroneous execution. Hence the presence of this code hazard is an attribute which indirectly enhances the software detection capability.

8.9. Conclusion

A significant amount of software produced for microprocessor applications is written in a high-level language which is independent of the target processor. This software is converted by a translator to equivalent machine code implementation. The programmer, therefore, does not have any control over the nature of the machine code generated. This code could contain embedded hazards which cause catastrophic failures during erroneous execution.

It has been shown that translators can implement techniques to manipulate machine code. Such manipulation can prevent the release of hazardous code by the

translator. The techniques involve encouraging and discouraging particular translator practices. Within program areas instructions are selected with respect to their opcode and operand. Both may have a hazardous format. Equivalent operations can be chosen to provide alternative opcodes, and operands can be manipulated by implementing different addressing modes so that no critical hazards are generated. In addition, the configuration of the address space can be altered to cover certain hazards. The manipulation techniques proposed to prevent hazardous code generation in the data and input/ output reserved areas are dependent to a large extent on the data structures implemented by the microprocessor. Applying the collection of manipulation techniques can prevent the generation of critical hazards.

The proposed translator techniques are influenced by the format and content of the target processor instruction set. General inferences are made for instruction set characteristics.

The ability to manipulate translation of target machine code also facilitates further working of the code to introduce a detection capability for erroneous execution. This is achieved by encouraging the generation of code with a restart instruction format. Such enhancement of the code can further improve the reliability of a microprocessor application by reducing error latency.

Chapter 9

MICROPROCESSOR DESIGN FOR FAULT TOLERANCE

CHAPTER NINE

MICROPROCESSOR DESIGN FOR FAULT TOLERANCE

9.1. Introduction

The hazard of erroneous microprocessor behaviour is modelled and evaluated in the early chapters of this thesis. A technique is proposed in Chapter 5 using software implemented fault tolerance to provide detection of erroneous execution. A utility has been built which automatically applies the proposed software based technique to target code, with the performance of the enhanced code being analysed and the benefit of its improved detection capability demonstrated. Furthermore, the previous chapter discusses methods of software translation which reduce the hazard of the target source code in respect of erroneous execution. This chapter concludes the research presented in this thesis by considering design features that can be incorporated within the architecture of a microprocessor to provide fault tolerance.

The design features presented identify particular characteristics of erroneous execution. Once erroneous execution is distinguished from valid processing, appropriate recovery action can be initiated to restore the integrity of the microprocessor system. The performance of the implemented design features and their attributed architecture overhead for a microprocessor is discussed.

9.2. The Effectiveness of Fault Tolerance

As discussed in Chapter 1, the implementation of fault tolerance at the logic level within a microprocessor's architecture incurs large overheads, in particular, a large extension in the number of gates is required. Chakraborty & Ghosh [1988] show that logic faults have a good correlation to functional errors in processor operation. It is therefore feasible to implement techniques facilitating functional-level fault tolerance. These techniques require less gate overhead than logic fault tolerance because they only attempt to detect particular characteristics of erroneous behaviour rather than all possible failure patterns.

Established functional-level fault tolerant techniques for microprocessor systems, called capability checks, are discussed in Chapter 2. Application of these techniques to microprocessor systems involves the addition of dedicated hardware units and/or software manipulation. Fault tolerant techniques which operate on functional aspects of erroneous execution are extremely difficult to assess. Erroneous behaviour will vary from application to application, depending on the target software and processor and hence the detection capability of the applied technique will vary too. This is known as the 'instruction sequence' dependency. Nevertheless, inclusion of capability checks within microprocessor systems can be beneficial where high reliability is required.

9.3. Implementing Fault Tolerance

The implementation of fault tolerance as described in Chapter 2 involves the co-ordinated detection of an error and restoration of the microprocessor system integrity. This thesis considers the first stage of fault tolerance, i.e. the detection of erroneous microprocessor behaviour as described in Chapter 3. It is proposed that detection can be achieved through the generation of a restart outcome during erroneous execution. Such a processing outcome re-establishes control of the program flow by directing execution to a predefined location in the address space. In order to complete recovery, a restart outcome must initiate execution of a module of code that restores system integrity. The predefined location in memory, accessed as a result of a restart outcome, is therefore hardwired within the microprocessor architecture to be a section of Read Only Memory (ROM). The micro-code in the ROM is programmed with a recovery routine for the application software.

Microprocessor design features are proposed in this chapter which facilitate an increased probability of erroneous execution generating a restart outcome, without the need for peripheral circuitry or software manipulation.

9.4. Influences on Microprocessor Design

It is very difficult to present a list of influences on microprocessor architecture design because of the diversity of application requirements between different processors. For instance, the Intel 4004 is a 4-bit microprocessor designed for simple arithmetical

functions, while the Motorola 68(7)05 is an 8-bit processor specifically designed for programmable controllers, and the Advanced Micro Device Am29000 processor is a 32-bit processor designed for real-time systems requiring large amounts of processing. The architecture of these microprocessors satisfies many design requirements including processing performance, processing capability, and cost. This is achieved through the implementation of various design features: for example, the AMD Am29000 implements a Reduced Instruction Set Computer (RISC) architecture incorporating a cache memory, :struction pipeline, and reduced instruction set in order to improve its run-time processing capability.

The following sections of this chapter consider design features which can be incorporated into the microprocessor architecture to facilitate the self-detection of erroneous behaviour. In particular, design features are proposed which facilitate detection of the erroneous execution modelled in Chapter 3, and the hardware representation of the software implemented fault tolerant technique presented in Chapter 5.

9.5. Instruction Set Architecture

All programs which are executable on a target microprocessor consist of a sequence of machine instructions. Typical instruction formats contain the following basic elements:

- an *opcode* specifying the operation,
- addressing mode specification for each of the input operands and result, and
- addressing mode data, e.g. immediate data for direct addressing.

An instruction's addressing mode is designated by either the opcode or a reserved tag within the operand. The Motorola 68000 processor family defines the addressing mode of instruction operands within the opcode, whilst the Intel 8086 processor implements a tag within the operand bit format which is decoded to ascertain the operand's addressing mode.

The architecture and organization of the processor instruction set can be manipulated to enhance the inherent detection capability of erroneous execution by the microprocessor. Techniques considered within this section involve the instruction set mix, the opcode map, and the instruction operand requirements.

9.5.1. Instruction Set Mix

The selection of instruction operations for inclusion in an instruction set is the subject of much research [Tanenbaum, 1990]. Instructions can be separated into two broad groups: general purpose and specialized instructions. General purpose instructions include data movement operations that are needed by almost every application. Specialized instructions are only useful in specific applications, e.g. the Motorola 68000 instruction MOVEP takes the content of the D register and stores it in alternate bytes and is intended for communication with specific 8-bit peripheral devices.

Studies of instruction set usage have led to the development of Reduced Instruction Set Computers (RISC) such as the Advanced Micro Device Am29000 processor. These processors incorporate only the most used and general purpose instructions unlike Complex Instruction Set Computers (CISC) which have large instruction sets that are often extended when the processor is upgraded through an upwardly compatible revision. Within the RISC processor, particular tasks that can be achieved by a single sophisticated CISC instruction may require several of the more basic instructions provided in its instruction set. Although the application software for a particular task is larger on a RISC compared to a CISC processor, its simpler data path processing (partially facilitated by the RISC instruction set) improves overall performance.

The instruction mix analysis of Chapter 4 which is used to evaluate the model of erroneous microprocessor behaviour highlights the advantages and disadvantages associated with the inclusion of restart and stop/wait outcome generating operations within the processor instruction set respectively. Stop/wait outcome instructions initiate catastrophic failure during erroneous execution because all independent operation is lost, and restart outcome instructions provide a controlled route for erroneous

execution to a recovery routine which restores processor integrity. Obviously the number of stop/wait outcome generating instructions in the instruction set should be minimized, although the degree of their influence on erroneous execution is dependent on their activation rather than their occurrence in an instruction mix.

9.5.2. Opcode Maps

An instruction opcode is designed to have sufficient bits to identify all facilitated unique operations. The number of bits, n, used to specify an opcode format should be an integral multiple, m, of the processors memory element length b (usually a byte) in order to simplify data flow processing. Additionally, the number of bits specified for an opcode should be kept as small as possible in order to reduce the memory requirement of software. Hence opcodes which specify i instruction operations are designed with a bit length n where,

$$i \leq 2^n. \tag{9.1.}$$

Re-arranging gives,

$$n \geq \frac{log\ i}{log\ 2}, \tag{9.2.}$$

$$n \geq log_2\ i, \tag{9.3.}$$

and,

$$n = m.b. \tag{9.4.}$$

The specified opcodes form an instruction set, and the range of opcode formats is usually referred to as the processors opcode map.

Equation (9.1.) notes that there may be some redundancy in the opcode map. This redundancy is acceptable in many architectures because the processing benefits of fixed-length instruction implementation within the overall processor design outweigh the sacrifice of larger average code size [Hennessy & Patterson, 1990].

All redundant opcode formats in the opcode map should be defined to generate a restart outcome. In this manner, execution of an undeclared operation, which is synonymous with erroneous microprocessor behaviour, is detected.

The opcode map can also be designed to eliminate the hazard of instruction operands associated with a local displacement by a relative branch. Program flow typically follows the 'principle of locality', well known by memory systems' designers, and can be explained by using a rule of thumb:

> 'A program executes about 90% of its instructions in about 10% of its code'.

The implication is that the majority of program flow involves branches whose target destination is in the local vicinity to the location of the generating instruction in the program code. Hennessy & Patterson [1990] describe the analysis of benchmark programs which support this assertion.

Many processor architectures define local branch displacement information to be contained with a branch instruction operands. These operands are susceptible to interpretation as an opcode during periods of erroneous execution. It is therefore proposed that the processor instruction set reserves areas representing low and high values in the opcode map for restart generating instructions, see Figure 9.1. Such opcode map organization ensures that local branch instruction operands interpreted as an opcode generate a restart outcome and hence detect erroneous execution.

A particular hazard associated with branch locality is observed within the Motorola 68(7)05 microprocessor instruction set. Here, branch instructions specify their displacement in the attached operands. Low order relative displacements for forward branch instructions which specify byte operands with the most significant four bits set to '0000' or '0010' are potential invalid branches.

The Motorola 68000 microprocessor reserves high value opcode formats for future upwardly compatible processor extensions. Execution of a memory element with such an opcode format results in a restart outcome via a software exception. This attribute

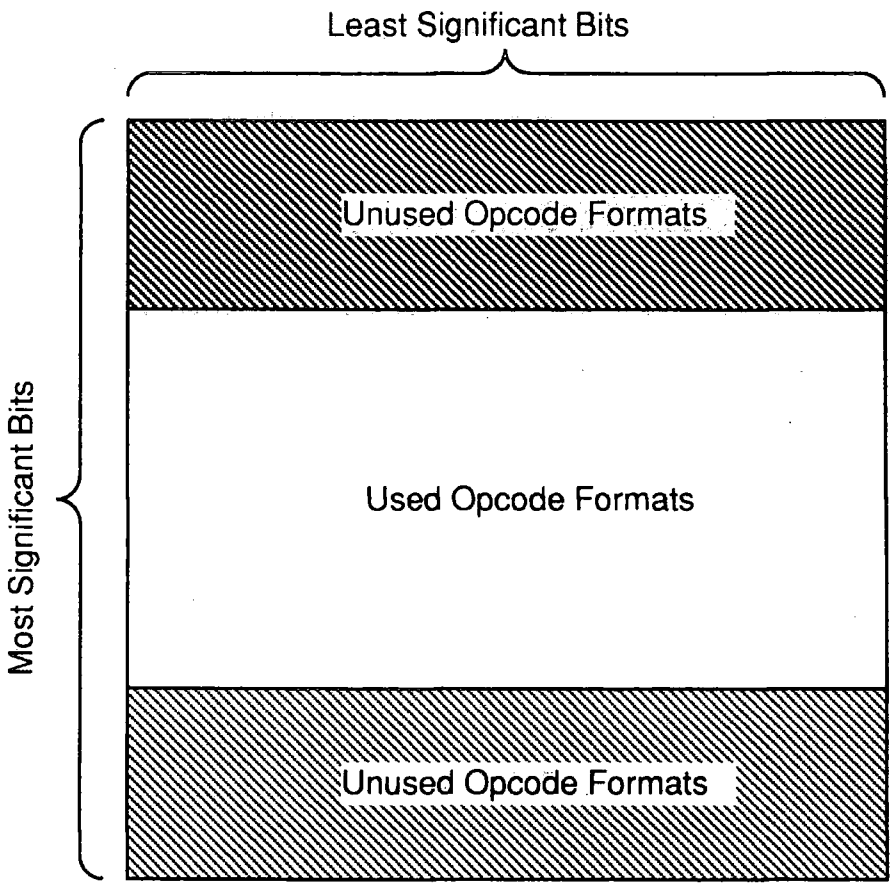


Figure 9.1. : Microprocessor Opcode Map

The unused opcode bit formats represent high and low values. These opcodes can be set to generate a restart outcome. Relative branch instructions specify a local displacement. Operands containing such displacements which are erroneously interpreted as an opcode will now generate a restart outcome and hence detect erroneous execution.

of the Motorola 68000 instruction set facilitates a detection capability where relative branch instructions specify an operand to contain a local backward displacement.

9.5.3. Operand Requirements and Specification

Operands have a bit size equivalent to an opcode in order to preserve the memory and data path organization associated with opcode processing. Microprocessor architects aim to keep the number of operands as small as possible in order to reduce instruction decoding and hence improve performance [Ciminiera & Valenzano, 1987]. Additionally, implementing fewer operands reduces the memory requirement of a program. The number of operands required depends on the amount of data to be processed and the addressing mode used.

The code extension required by the application of the software implemented fault tolerant techniques proposed in Chapter 5 is largely influenced by the operand requirements of instructions within the processor instruction set. Particular influence is observed where default size mechanisms are inserted with target software; the default size is equivalent to the maximum number of operands required by an instruction in the instruction set. Optimum size detection mechanisms may require less additional code, depending on the placement conditions at each insertion. Further details of detection mechanism construction can be found in Chapter 5. In order to reduce the code extension required when applying the software implemented fault tolerant technique proposed in Chapter 5, the instruction architecture should specify instructions to have fewer operands.

Register oriented addressing modes can be encouraged to reduce the operand requirements of an instruction. Such addressing modes remove the necessity for operand specification of absolute addresses or immediate data. Details of the registers to be processed can be specified within the opcode bit format. Where this information cannot be incorporated within the opcode bit format, a single operand can be specified to contain the register allocation information. The bit format of the operand is designated to represent a restart generating opcode. In many instances a single operand will be shorter than the number of operands required to represent an absolute address.

9.6. Input/Output Communication Ports

Chapter 5 discusses two attributes of a microprocessor's architecture which facilitate an erroneous execution detection capability for the reserved input/output port locations in the address space. These attributes can be incorporated into the design specification of a processor.

Firstly, instructions that communicate with external devices may specify access to an input/output port within an operand. In particular, individual instruction operands may be set to contain the whole or partial address of the communication port. The location of these communication ports in the memory map can be defined to take bit formats which represent restart generating instructions in the opcode map. Hence, erroneous execution of an input/output port address as an opcode results in detection and recovery can be initiated.

Secondly, erroneous execution may itself interpret an input/output port location as an opcode. The microprocessor design can incorporate the hardwiring of particular bits in the input/output port such that the bit format represents a restart instruction, as Figure 9.2. Erroneous execution of this location as an opcode is self-detected. This method, of course, incurs an overhead in that the data transfer capability to an external device is reduced because of the redundant bits reserved for erroneous execution detection.

9.7. Memory Organization

This section discusses techniques which involve the organization and implementation of memory used by a microprocessor.

9.7.1. Memory Alignment

Some microprocessors require elements (e.g. byte, double-byte, or quad-byte) in memory to be aligned according to the memory organization. A memory element of size s bytes resident at location A_d in the address space is aligned when,

$$[A_d \bmod s] = 0 \tag{9.5.}$$

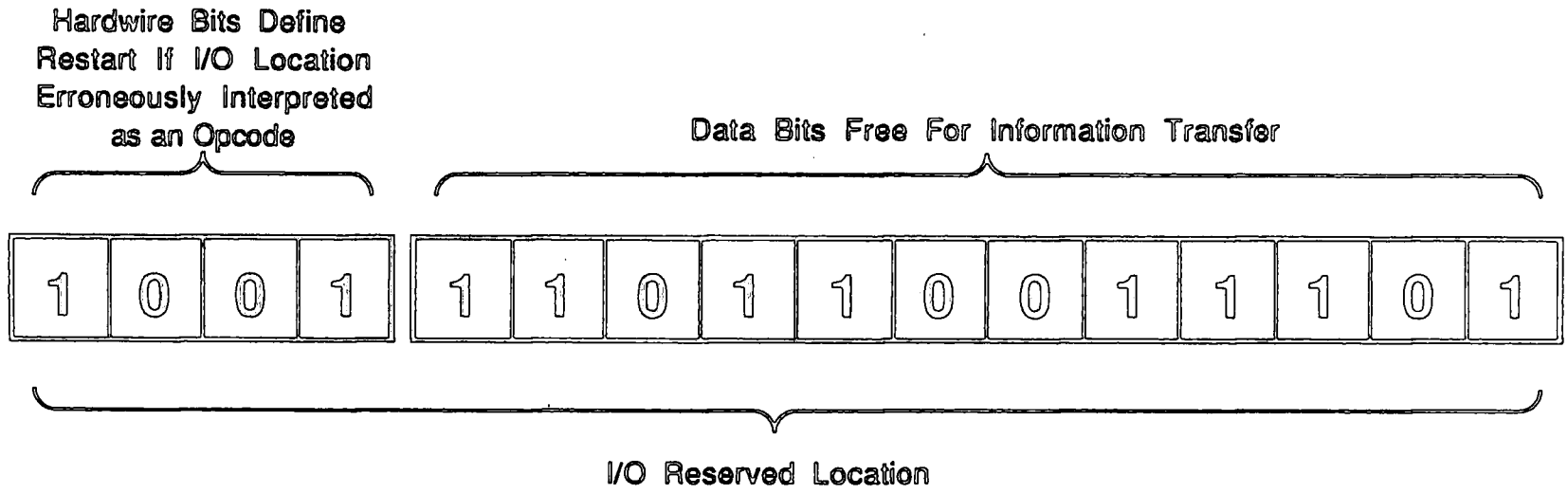


Figure 9.2. : Input/Output Location Content

Hence, byte memory elements in a byte orientated memory organization will always be aligned. Byte memory elements in a double-byte orientated memory are aligned on even byte boundary addresses, but mis-aligned on odd byte boundary addresses. Mis-aligned element access specified by an instruction requires multiple physical memory accesses, whilst aligned memory access requires only one physical memory access by the microprocessor hardware. From a performance perspective, therefore, elements of memory should be aligned rather than mis-aligned.

The memory organization employed in a microprocessor architecture has particular implications for the software implemented fault tolerant techniques proposed in this thesis. Those processors that implement a byte memory organization for the program area allow any instruction operand to be processed erroneously as an opcode. Hence erroneous execution in the program area is not hindered by the memory organization. Other means of memory organizations allow the possibility of mis-aligned opcode access. Mis-aligned opcode access is a characteristic of erroneous behaviour. Fault tolerant techniques can be implemented to detect this error.

The Advanced Micro Device Am29000 microprocessor and the Motorola 68000 microprocessor family implement a similar approach to mis-aligned opcode access for their respective quad-byte and double-byte memory organization. Access to mis-aligned double-byte elements in memory within the Motorola 68000 microprocessor architecture generates an exception which, naturally, is called the 'odd byte address' exception. This method of imposing a double-byte organization in memory to improve the operational performance of the processor, also makes possible a detection capability to prevent erroneous execution. The Advanced Micro Device Am29000 processor has a similar function. Its exception generation can, however, be dis-abled by a special status register. Enabling the mis-alignment exception results in the least two significant bits of the program counter being masked and hence an opcode address is generated. Erroneous execution is, therefore, re-synchronized.

The design feature described in this section can be incorporated into the architecture of a microprocessor. Digital circuitry can generate a restart outcome when

it detects mis-aligned memory access. Such an outcome can be used to initiate a recovery procedure and hence improve the reliability of the microprocessor system.

9.7.2. Defining Memory Utilization

The address space of a microprocessor can be divided according to its functional allocation. Chapter 5 initially divides the address space into used and unused areas discusses the implications of the division with respect to erroneous behaviour. Moreover, the said chapter describes the digital implementation of a hardware unit called an Access Guardian which detects the unused area access characteristic of erroneous behaviour. The design of the Access Guardian is not complex and can be incorporated into the architecture of a microprocessor.

There, nevertheless, remains the problem of detecting erroneous execution in the used area of the address space. Glaser & Masson [1982] present a hardware unit referred to as a 'SAFE ROM' which has been implemented within a microprocessor system by Li et al [1984]. The SAFE ROM is a one-bit wide memory which is attached to each memory element (in this case byte) of Read Only Memory (ROM) to signify its usage as either opcode or operand, see Figure 9.3. Invalid opcode address access is identified by detection circuitry which determines whether or not the location in question has its SAFE ROM bit set to opcode or operand.

A similar approach is proposed here which can be applied to all address space locations regardless of their implementation in either Read Only Memory or Random Access Memory (RAM). Instead of implementing additional memory, a bit of each memory element is reserved to perform the SAFE ROM function. Hence, there is not a memory overhead as such but rather memory redundancy associated with the reserved bit, see Figure 9.4. This method provides detection of erroneous execution for all physically implemented memory. The memory redundancy incurred by applying this technique can be calculated, thus:

$$\text{Memory Redundancy } (\%) = \frac{1}{n}, \tag{9.6.}$$

where the opcode memory element has n bits.

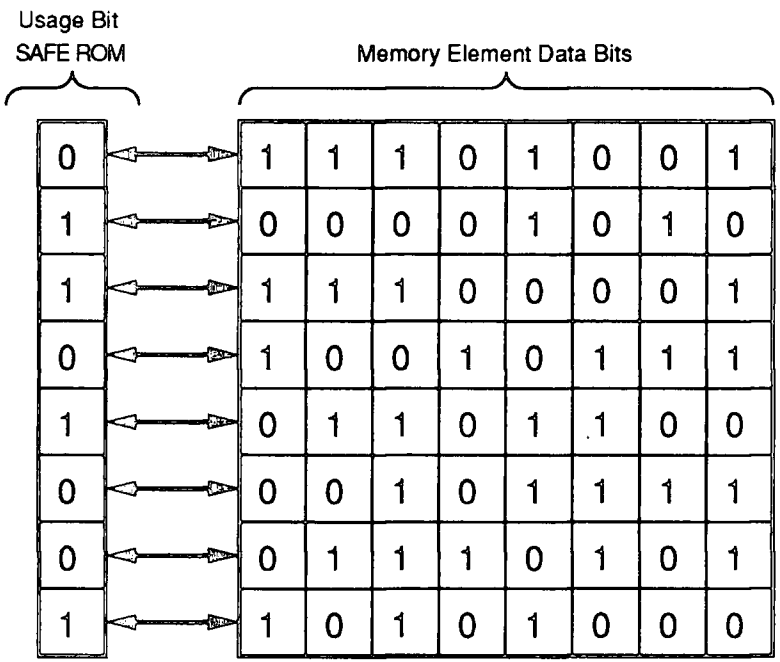


Figure 9.3. : SAFE ROM [Glaser & Masson, 1982]

Usage Bit = 1 : Defines Opcode
Usage Bit = 0 : Defines Operand

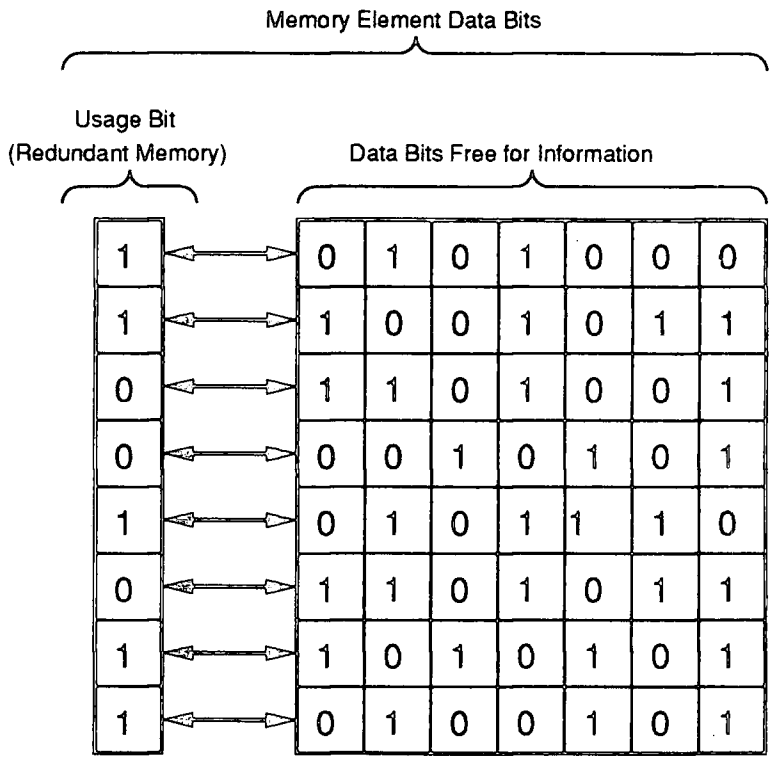


Figure 9.4. : Memory with Utilisation Assignment

Usage Bit = 1 : Defines Opcode
Usage Bit = 0 : Defines Operand

The memory redundancy associated with this technique may be considered significant for some processor applications. For example, microprocessors whose memories are organized as byte or double-byte structures have a 12.5% and 6.25% redundancy in their memory respectively. Nevertheless, the technique is extremely effective, providing detection of erroneous execution except in circumstances involving re-synchronization. Detection capability is important for systems requiring high reliability and in these applications the memory redundancy is expected to be acceptable.

9.8. Monitoring Branch Activity

The architecture of a microprocessor can be extended to incorporate design features facilitating the recognition of branch operations similar to that provided by the Motorola 68030 microprocessor. Such a facility could be used to activate special circuitry dedicated to determining whether an invalid or valid branch is being processed.

Two methods of determining invalid branch activity are suggested here. Firstly, the software implemented fault tolerant technique proposed in Chapter 5 can be incorporated into the microprocessor architecture. The required digital circuitry is based on a logical AND function using, as inputs, the recognition of branch activity and the opcode 'usage' bit proposed in the previous section. The outcome is detection of all invalid branches regardless of whether or not their destination leads to the re-synchronization of erroneous execution.

The second technique is based on verifying branch activity, and is commonly referred to as 'signature analysis'. The theory supporting the technique is described in Chapter 2. Schuette & Shen [1986] have implemented the technique using additional digital circuitry in a Motorola 68000 microprocessor based system, which incurred a 17% gate overhead compared with the number of gates in the processor. More recently, the technique has been incorporated within the architecture of development processors where it had a 13% chip area overhead [Leveugle et al, 1990]. The technique detects erroneous execution by failing to verify the occurrence of a valid branch.

On identifying an invalid branch erroneous execution is detected and appropriate hardwired recovery action can be initiated. Although the techniques described above are only activated when branch activity is recognized, they do detect re-synchronized erroneous execution. This contrasts with the technique based on defining memory utilization which is activated more regularly because of its on-line monitoring process, but which cannot detect re-synchronized erroneous execution.

9.9. Conclusion

Design features which can be incorporated into the architecture of a microprocessor to provide a self-detection capability for erroneous behaviour have been proposed. These include suggestions for the instruction set architecture and memory organization. The benefit of the inclusion of these techniques in the microprocessor hardware (enhanced detection capability) are dependent on the particular instruction sequences of erroneous execution for a target processor system. Nevertheless, capability checks are being included into microprocessor designs, notably the mis-aligned opcode address exception. Further capability checks, including those proposed in this chapter, may be implemented within commercial microprocessor designs in the future.

Chapter 10

CONCLUSION

CHAPTER TEN

CONCLUSION

10.1. Microprocessor Controllers For Industrial Applications

In recent years it has become popular practice to implement industrial control systems using digital circuitry with an embedded microprocessor rather than analogue systems. Microprocessor controllers provide a flexible design approach, the nature of their operation being easily tailored to particular tasks through the alteration of control software. Digital systems, however, are more susceptible to transient disturbances, common within industrial environments, than similar analogue systems. Analogue systems tend to pass the effects of a transient disturbance as a temporary processing discrepancy, whilst digital systems, because of their discrete state nature, can have their operation disrupted. This thesis addresses the problem of improving the reliability of low budget microprocessor systems where fault masking is considered too expensive.

10.2. Reliable Microprocessor Controllers

The failure process of a digital system involves the manifestation of a logic fault, its activation as an error, and finally error spawn until a fatal operating condition is generated. Faults can be manifested as either temporary or permanent logic corruption. Temporary faults have been observed to cause a significant proportion of microprocessor system failures. Studies, reviewed in Chapter 2, suggest that in excess of 90% of processor failures are generated by temporary faults rather than permanent faults.

Temporary faults, attributed to disturbances in the operating environment of a microprocessor based controller, are called transient faults. Environmental disturbances can involve electro-magnetic interference (EMI), electro-static discharge (ESD), electronic noise, ionizing radiation, and power supply fluctuations. The operating environment can often be harsh. In order to achieve high reliability,

microprocessor controllers can implement fault tolerance. Fault tolerant techniques implement four tasks in sequence:

- error detection,
- damage assessment,
- error recovery, and
- system restoration.

Although this thesis focuses on error detection, the remaining fault tolerant tasks are equally important and should be considered when implementing a fault tolerant system.

Additional circuitry to detect individual logic faults can be prohibitive within a low budget microprocessor architecture. An alternative technique for detecting logic faults is based on the premise that logic faults are complemented by processing errors. The technique is referred to as functional fault tolerance because it relies on distinguishing between valid and invalid characteristics of microprocessor execution.

Functional fault tolerance must ensure that erroneous microprocessor execution does not generate a fatal error and hence catastrophic failure. To achieve this, detection techniques can be applied to recognize different attributes of erroneous execution. These techniques are collectively known as 'capability checks'. In order to design and evaluate the effectiveness of capability checks it is necessary to model erroneous microprocessor behaviour.

10.3. Modelling Erroneous Microprocessor Behaviour

Erroneous microprocessor behaviour involves either erroneous data or program flow within executing software. The 'reasonableness' of data flow can be checked by the operating software. Erroneous program flow, however, cannot be verified in this manner because predictable operation of the software is lost. This has serious implications for industrial applications where microprocessors are responsible for the monitoring, process, or control of equipment. Unpredictable processor action can command equipment to malfunction, the hazard of this situation being dependent on the equipment's task.

A model has been developed which investigates the program flow associated with erroneous microprocessor behaviour. Erroneous execution is defined to be initiated by a temporary fault generating an Initial Erroneous Jump (IEJ) through corruption of the processor's program counter. Ensuing erroneous microprocessor behaviour is characterized by periods of linear erroneous execution interspersed with further erroneous jumps called Subsequent Erroneous Jumps (SEJs). In addition, the model allows consideration of particular processing outcomes associated with catastrophic failure and recovery. Recovery is achieved through the processing of an instruction developing a restart outcome, which directs execution to a pre-defined location in memory where a recovery routine resides. The recovery routine is programmed to fulfil the restoration requirements of the application software, two possible recovery strategies are reset and roll-back.

The model of erroneous behaviour is applied to a selection of 8, 16, and 32-bit processors. The following microprocessors are assessed: (8-bit) Motorola 6800, Intel 8048, and Intel 8085; (16-bit) Intel 8086, Motorola 68000, and Motorola 68010; (32-bit) Advanced Micro Device Am29000, Motorola 68020, and Intel 80386. All processors are assumed to have a random content address space. Erroneous execution is evaluated using instruction mix analysis to predict the mean expected operation.

The character of erroneous jumps is studied. The model assumes that IEJs have a random target in the address space. Such erroneous jumps have particular significance in relation to microprocessor reliability when their destination is in the unused area whose code attributes are unknown. Where an IEJ directs erroneous execution to the used area, further erroneous jumps (SEJs) occur as a result of invalid software processing. The model suggests that SEJs typically generate a local target and hence erroneous execution initiated within the used area is likely to remain there for a significant period. This is a hazardous situation because not only is the processor out of control but its erroneous activity could be mutilating system integrity, making restoration of the microprocessor system more difficult.

A Markov process is used to predict the reliability of a microprocessor. In particular, the Mean Time To Failure (MTTF) parameter is used because it has more intu-

itive meaning to reliability engineers. The instruction mix analysis for the selection of microprocessors described above suggests that both the instruction distribution and the processor architecture can have a significant influence on reliability. The Motorola 68000 microprocessor family have the highest reliability with an MTTF three times the mean inter-arrival time of events that initiate erroneous execution. These processors have instruction sets which define the execution of an undeclared opcode and mis-aligned memory access to generate a restart outcome. The Advanced Micro Device Am29000 processor has an MTTF prediction twice that of the mean inter-arrival event period which is due solely to undeclared opcodes generating a restart outcome. The remaining microprocessor reliability models predict a MTTF of similar magnitude to the mean inter-arrival event period. These processors do not define their undeclared opcodes to have a restart outcome, and implement a byte memory organization and hence mis-aligned memory access is impossible.

The availability of a microprocessor system is dependent on the detection of an error and time taken to restore the system integrity. A general model is presented: the influences on availability are the mean inter-arrival event period, the processor operational frequency, and the size of the routine required to restore system integrity. The latter two factors are processor and application dependent. Microprocessor architectures that require little restoration activity and simple application software can reduce the size of the recovery routine. These, together with high processor operating speeds, facilitate higher availability.

10.4. Detecting Erroneous Microprocessor Execution

This thesis proposes a new capability check for detecting erroneous microprocessor execution. The technique is based on software implemented fault tolerance.

The aim of the technique is to identify, through static analysis, the potential targets of erroneous jumps, referred to as invalid branches, and to place at these locations a software detection mechanism which is activated by the erroneous execution. Invalid branches are unsynchronized erroneous jumps and should not be confused with synchronized erroneous jumps. Interpretation of a memory location containing an

opcode is termed 'synchronized', whilst interpretation of any other content is termed 'unsynchronized'.

Different approaches are required when applying software implemented fault tolerance to the used and unused areas of the address space respectively. The first approach considers the unused area, which can consist of physical and non-existent memory. Physical memory is filled with restart generating instructions which do not have an operand requirement. In this way, erroneous execution at any location will develop a restart outcome and hence detection of erroneous behaviour. Non-existent memory requires a hardware solution, and hence a unit called an Access Guardian is designed which detects memory access by monitoring the processor address bus. The complexity of the Access Guardian is dependent on the contiguity of non-existent memory locations, and whether or not the processor has a multiplexed address/data bus such as the Intel 8086 microprocessor.

Secondly, within the used area, detection mechanisms are inserted within the software at invalid branch targets. Some manipulation of the application software may be necessary so that the its function is not disturbed by the placement of detection mechanisms. Construction principles for the detection mechanisms and an algorithm for their placement within the application program are presented. An important limitation of the technique is that of placement deadlock. This describes a situation where an invalid branch destination cannot be covered by a detection mechanism placement because the generator and destination of the invalid branch are both within the same instruction, or the destination resides at the location immediately following the generating instruction. The former can generate a catastrophic failure if an infinite execution loop is created.

A software tool, called the Post-programming Automated Recovery UTility (PARUT), has been developed as a prototype in order to assess the capability of the software implemented fault tolerant technique and to assess the feasibility of developing a standard software tool to apply the technique. The structure and organization of the prototype is described. The tool is designed to be robust, capable of generating enhanced program code for a variety of target processors. The future of

PARUT appears to be its incorporation, as an processing option, within a translator. This is because PARUT uses much of the information inherently required within the translation process, and its phase of activity immediately follows translation.

10.5. Evaluating Fault Tolerance

The dynamics of erroneous execution can only be evaluated through instruction sequence analysis which involves tracing the execution attributed with each initiated period of erroneous behaviour.

The effectiveness of the software implemented fault tolerant technique proposed in this thesis is evaluated by investigating the erroneous behaviour of application software before and after application of the technique. Erroneous microprocessor behaviour is investigated using fault emulation and fault injection experiments. The fault injection experiment involves physically inserting faults on the the address bus, data bus, and program counter during instruction and data cycles. The fault emulation experiment involved inserting faults within a register model of a processor. Almost 4000 faults are investigated for a selection of three programs, each with a different application processor.

Improved performance is observed in the processor systems when they employ the software implemented fault tolerant technique. The degree of improvement is related to the number of detection mechanism placements in the application software. The effectiveness of the software technique is clearly demonstrated. The memory overhead and performance of the software technique is, however, application specific. Within the example programs, the application of the technique required an approximate software extension of 20% to 30% for 16-bit and 32-bit microprocessors.

10.6. Generating Non-Hazardous Software

Many microprocessor systems utilize application code written in a high level language which is independent of the target processor architecture, e.g. the programming language C. In these instances a translator is used to convert the source code through levels of abstraction to the object code. This process can be influenced so that target

code is produced without the hazards associated with catastrophic failure, and with a high inherent detection capability against erroneous execution.

Five tasks of the translation process are identified which can influence the production reliable code. Firstly, opcodes with more than one addressable memory element can be hazardous on a mis-aligned opcode access. The selection of opcodes during translation should avoid identified hazardous opcodes, their function being implemented other equivalent instructions. Secondly, the addressing mode selected for an instruction should not generate hazardous operands. Thirdly, macros used to implement high level language constructs should not incorporate instruction sequences with a hazard. Fourthly, peephole optimization should not create new code hazards. Finally, address space allocation for the object code should not introduce hazards, for instance through relative address operands.

These translator proposals have not been implemented. An alternative method of generating non-hazardous software is to design a microprocessor architecture which inherently defines code with a detection capability against erroneous execution.

10.7. Microprocessor Design for Fault Tolerance

Fault tolerant techniques implemented as additional hardware circuitry, with or without software manipulation, can be incorporated within the architecture of a microprocessor. Many modern microprocessors incorporate a mis-aligned memory access exception, and other prototype processors implement signature analysis. The techniques proposed in this thesis concerning software implemented fault tolerance can also be embedded within the design of a microprocessor. These techniques use an Access Guardian to detect all unused area access, and software detection mechanisms to detect invalid branches regardless of their re-synchronization. Additional techniques involve influencing the design of the processor instruction opcode map and operand requirements, reserved input/output ports, and memory organization.

10.8. Summary

Microprocessor systems are incorporated within many industrial control systems. Such applications are often required to be highly reliable. Working environments can,

however, be harsh and microprocessor systems are prone to disruption from transient disturbances. It is therefore necessary to apply fault tolerance to the microprocessor system in order to improve its reliability. The sophistication of the fault tolerance may be limited by budget constraints which prevent fault masking.

The solution is the application of capability checks within a uniprocessor controller. The capability checks identify particular characteristics of erroneous processor behaviour and initiate recovery. This thesis models erroneous microprocessor behaviour and proposes a new low-cost software-implemented capability check involving the recognition of invalid branches. The effectiveness of applying the capability check is demonstrated; however, a general assessment cannot be made because the techniques action is application specific. The error detection capability can be further improved by strategically selecting several capability checks for collective application. It should be realized that these techniques cannot guarantee enhanced reliability because they are reliant on particular attributes of erroneous execution being exhibited. It is pertinent for the reliability engineer to incorporate a back-up detection facility into the system design, such as a watchdog timer as well as a fail-safe action, in order to prevent unpredicted failure modes causing a catastrophic outcome.

Bibliography and References

Bibliography

[British Telecom, 1986]
British Telecom, *Handbook of Reliability Data for Components Used in Telecommunication Systems (HRD4).*, 1986.

[Fontaine & Barrand, 1989]
Fontaine, A.B. & Barrand, I., *80286 and 80386 Microprocessors : New PC Architectures.*, Macmillan Education, 1989.

[Intel, 1982]
Intel, *8086/8088/8087/80186/80188 Programmer's Pocket Reference Guide.*, Intel Corporation, 1982.

[Intel, 1988]
Intel, *80386 Programmer's Reference Manual.*, Intel Corporation, 1988.

[Johnson, 1989]
Johnson, M. (ed), *Am29000 Users Manual.*, Advanced Micro Devices, 1989.

[Klaassen & van Peppen, 1989]
Klaassen, K.B. & van Peppen, J.C.L., *System Reliability : Concepts and Applications.*, Hodder & Stoughton (Edward Arnold Division), 1989.

[Lewis, 1987]
Lewis, E.E., *System Safety Analysis : Human Error.*, in 'Introduction to Reliability Engineering', John Wiley & Sons, New York, 1987.

[Motorola, 1983]
Motorola, *M6805 HMOS, M146805 CMOS Family : Microcomputer/ Microprocessor User's Manual.*, Motorola Inc., 1983.

[Motorola, 1984]
Motorola, *M68000 16/32-Bit Microprocessor : Programmer's Reference Manual.*, Motorola Inc., 1984.

[Motorola, 1987]
Motorola, *MC68030 Enhanced 32-Bit Microprocessor User's Manual.*, Motorola Inc., 1987.

References

[Aho & Ullman, 1977]
Aho, A.V. & Ullman, J.D., *Principles of Compiler Design.*, Addison-Wesley Publishing Company, 1977.

[Amerasekera & Campbell, 1987]
Amerasekera, E.A., & Campbell, D.S., *Failure Mechanisms in Semiconductor Devices.*, John Wiley & Sons, 1987.

[Anderson & Lee, 1982]
Anderson, T., & Lee, P.A., *Fault Tolerant Terminology Proposals.*, Proc. FTCS-12, 1982, pp 29-33.

[Arlat et al, 1990]
Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E., & Powell, D., *Fault Injection for Dependability Validation : A Methodology and Some Applications.*, IEEE Trans. Soft. Engineering, Vol. 16, No. 2, 1990, pp 166-181.

[Armstrong & Devlin, 1981]
Armstrong, J.R., & Devlin, D.E., *GSP - A Logic Simulator for LSI.*, Proc. 18th Annual Automation Conf., 1981, pp 518-524.

[Avizienis, 1976]
Avizienis, A., *Fault Tolerant Systems.*, IEEE Trans. Computing, 1976, pp 1304-1311.

[Ball & Hardie, 1969]
Ball, M., & Hardie, F., *Effects and Detection of Intermittent Failures in Digital Systems.*, AFIPS Proc. Fall Joint Computer Conf., Vol. 35, 1969, pp 329-335.

[Bennetts, 1979]
Bennetts, R.G., *Reliability of Engineering Systems.*, Lecture Notes (University of Southampton, England.), 1979.

[Bhar & McMahon, 1983]
Bhar, T.N., & McMahon, E.J., *Electronic Discharge Control.*, Hayden Book Company, 1983.

[Blough & Masson, 1987]
Blough, M.B. & Masson, G.M., *Performance Analysis of a Generalized Upset Detection Procedure.*, FTCS-17, 1987, pp 218-223.

[Bourne, 1982]
Bourne, S.R., *The UNIX System.*, Addison-Wesley Publishing Company, 1982.

[Carpenter, 1989]
Carpenter, G.F., *Database for Investigating the Effects of Induced Faults.*, Microprocessors and Microsystems, Vol. 13, No. 10, 1989, pp 627-636.

[Carter, 1985]
Carter, W.C., *Hardware Fault Tolerance.*, Resilient Computer Systems (ed. Anderson, T.), Collins Publishers, 1985, pp 11-63.

[Chakraborty & Ghosh, 1988]
Chakraborty, T., & Ghosh, S., *On Behaviour Fault Modelling for Combinational Digital Designs.*, Proc. Int. Test Conf., 1988, pp 593-600.

[Chen & Avizienis, 1978]
Chen, L., & Avizienis, A., *N-Version Programming : A Fault Tolerant Approach to Reliability of Software.*, Proc. Int. Symp. Fault-Tolerant Computing, 1978, pp 3-9.

[Chillarege & Bowen, 1989]
Chillarege, R., & Bowen, N.S., *Understanding Large System Failures - A Fault Injection Experiment..* Int. Symp. on Fault Tolerant Computing, 1989, pp 355-363.

[Choi et al, 1989]
Choi, G., Iyer, R., Saleh, R., & Carreno, V., *A Fault Behaviour Model for an Avionic Microprocessor : A Case Study.*, Proc. Int. Working Conf. on Dependable Computing for Critical Applications, 1989, pp 71-77.

[Ciminiera & Valemzano, 1987]
Ciminiera, L. & Valemzano, A., *Advanced Microprocessor Architectures.*, Addison-Wesley Publishing Company, 1987.

[Clark et al, 1987]
Clark, A.S., Jenkins, K., & Lipscombe, J.A., *Low-Cost Electronic District Control.*, Institution of Gas Engineers, 53rd Autumn Meeting, Communication 1353, 1987.

[Connet et al, 1972]
Connet, J.R., Pasternak, E.J., Wagner, B.D., *Software Defences in Real-Time Control Systems.*, Dig. Int. Symp. on Fault Tolerant Computing, 1972, pp 94-99.

[Cortes et al, 1986]
Cortes, M.L., McCluskey, E.J., Wagner, K.D., & Lu, D.J., *Modelling Power Supply Disturbances in Digital Circuits.*, IEEE Int. Solid State Circuit Conf. (Anahem, C.A.), 1986, pp 164-165.

[Crouzet & Decouty, 1982]
Crouzet, Y., & Decouty, B., *Measurement of Fault Detection Mechanisms Efficiency : Results.*, Proc. FTCS-12, 1982, pp 373-376.

[Cullyer, 1988]
Cullyer, W.J., *Implementing High Integrity Systems : the VIPER microprocessor.*, IEEE Computer Assurance Conference (COMPASS), 1988, pp 56-66.

[Cusick et al, 1985]
Cusick, J., Koga, R., Kolasinski, W.A., & King, C., *SEU Vulnerability of the Zilog Z-80 and NSC-800 Microprocessors.*, IEEE Trans. Nuclear Science, Vol. 32, No. 6, 1985, pp 4206-4211.

[Czeck & Siewiorek, 1990]
Czeck, E. W., & Siewiorek, D.P., *Effect of Transient Gate-Level Faults on Program Behaviour.*, FTCS-20, 1990, pp 236-243.

[Damm, 1988]
Damm, A., *Experimental Evaluation of Error Detection and Self-Checking Coverage of Components of a Distributed Real-Time System.*, Ph.D. Thesis, Technical University of Vienna, Austria, 1988.

[Dijkstra, 1972]
Dijkstra, *Notes on Structured Programming.*, 'Structured Programming', Academic Press Inc., New York, 1972.

[Duba & Iyer, 1988]
Duba, P., & Iyer, P.K., *Transient Fault Behaviour of a Microprocessor : A Case Study.*, Proc IEEE/ICCD, 1988, pp 272-276.

[Eckhardt & Lee, 1988]
Eckhardt, D.E. & Lee, L.D., *Fundamental Differences in the Reliability of N-Modular Redundancy and N-Version Programming.*, Journal of Systems and Software, Vol. 8, 1988, pp 313-318.

[Eifert & Schuette, 1984]
Eifert, J.B., & Schuette, J.P., *Processor Monitoring Using Asynchronous Signature Instruction Streams.*, Proc. FTCS, 1984, pp 394-399.

[Ferrara et al, 1989]
Ferrara, K.C., Keene, S.J., & Lane, C., *Software Reliability from a System Perspective.*, Proc. Annual Reliability and Maintainability Symposium, IEEE 1989, pp 332-336.

[Freer, 1987]
Freer, J.R., *System Design with Advanced Microprocessors.*, Pitman Publishers, 1987.

[Fujimura, 1989]
Fujimura, N., *Software Productivity in Built-In Microprocessors.*, Microprocessing and Microprogrammming, Vol. 28, 1989, pp 169-172.

[Fuller & Harbison, 1978]
Fuller, S.H. & Harbison, S.P., *The C.mmp Multiprocessor.*, Technical Report CMU-CS-78-146, Carnegie-Mellon University, October 1978.

[Glaser & Masson, 1982]
Glaser, R.E. & Masson, G.M., *The Containment Set Approach to Crash-Proof Microprocessor Controller Design.*, IEEE Trans. Computers, Vol. 31, No. 7, 1982, pp 689-692.

[Gunneflo et al, 1989]
Gunneflo, U., Karlsson, J, & Torin, J., *Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation.*, Int. Symp. on Fault Tolerant Computing, 1989, pp 340-347.

[Halse, 1984]
Halse, R.G., *Software Fault Tolerance for Small Digital Controllers.*, Ph.D. Thesis, University of Durham (England), 1984.

[Halse & Preece, 1985]
Halse, R.G., & Preece, C., *Erroneous Execution and Recovery in Microprocessor Systems.*, Software and Microsystems, Vol. 4, No. 3., 1985, pp 63-70.

[Halse & Preece, 1987]
Halse, R.G., & Preece, C., *Recovery Assessment After Microprocessor System Transient Disturbance.*, System Fault Diagnosis: Reliability and Related Knowledge-Based Approaches, (ed. Tzafestas, S.), Reidell Publishing Company, Vol. 2, 1987, pp 383-397.

[Hennessy & Patterson, 1990]
Hennessy, J.L. & Patterson, D.A., *Computer Architecture : A Quantitive Approach.*, Morgan-Kaufmann Publishers Inc., 1990.

[Hitt & Webb, 1985]
Hitt, E.F. & Webb, J.J., *A Fault Tolerant Software Strategy for Digital Systems.*, Proc. AIAA/IEEE 6th Digital Avionics Systems Conference, 1985, pp 211-216.

[Horowitz & Hill, 1986]
Horowitz, & Hill, *Eliminating Interference.*, 'The Art of Electronics', Cambridge University Press, 1986, p.307.

[IBM, 1986]
IBM, *Microprocessor Monitor and Reset Circuitry.*, IBM Technical Disclosure Bulletin (USA), Vol. 29, No. 2, 1986, p 611.

[IEEE, 1989]
IEEE, *Logic Comes to the Rescue.*, 'Faults and Failures', IEEE Spectrum, April 1989, p 18.

[IEEE, 1990]
IEEE, *What About Those Intel i486 Bugs?.*, 'Faults and Failures', IEEE Spectrum, April 1990, pp 8-9.

[Iyer & Rossetti, 1982]
Iyer, R.K. & Rossetti, D.J., *A Statistical Dependency of CPU Errors at SLAC.*, FTCS-12 (Santa Monica, CA.), 1982, pp 363-372.

[Iyer & Rossetti, 1986]
Iyer, R.K. & Rossetti, D.J., *A Measurement Based Model for Workload Dependence of CPU Errors.*, IEEE Trans. Computers, Vol. 35, No. 6, 1986, pp 511-519.

[Kim & Iyer, 1988]
Kim, S., & Iyer, R.K., *Impact of Device Level faults in a Digital Avionic Processor.*, AIAA/IEEE 8th Digital Avionic Systems Conf., 1988, pp 428-435.

[Kopetz, 1982]
Kopetz, H., *The Failure-Fault (FF) Model.*, Proc. FTCS-12, 1982, pp 14-17.

[Lala, 1983]
Lala, J., *Fault Detection and Reconfiguration in FTMP : Methods and Experimental Results.*, Proc. 5th IEEE/AIAA Digital Avionics Systems Conf. (DASC), 1983, pp 21.3.1-21.3.9.

[Lala, 1985]
Lala, P.K., *Fault Tolerant and Fault Testable Hardware Design.*, Prentice-Hall, 1985.

[Leveugle et al, 1990]
Leveugle, R., Michel, T. & Saucier, G., *Design of Microprocessor with Built-In On-Line Test.*, Proc. FTCS, 1990, pp 450-456.

[Li et al, 1984]
Li, K.W., Armstrong, J.R., & Tront, J.G., *An HDL Simulation of the Effects of Single Event Upsets on Microprocessor Program Flow.*, IEEE Trans. Nuclear Science, Vol. 31, No. 6, 1984, pp 1139-1144.

[Lin, 1988]
Lin, T-T, K., *Design and Evaluation of an On-Line Predictive Diagnostic System.*, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[Lin & Siewiorek, 1990]
Lin, T-T. Y. & Siewiorek, D.P., *Error Log Analysis : Statistical Modelling and Heuristic Trend Analysis.*, IEEE Trans. Reliability, Vol. 39, No. 4, 1990, pp 419-432.

[Littlewood, 1981]
Littlewood, B., *Stochastic Reliability Growth : A Model for Fault Removal in Computer-Programs and Hardware-Design.*, IEEE Trans. Reliability, Vol. 30, No. 4, 1981, pp 313-320.

[Liu, 1982]
Liu, B., *Soft Failure Detection and Correction in Microprocessor Characterization.*, Proc. FTCS-12, 1982, pp 458-460.

[Liu & Whalen, 1988]
Liu, K., & Whalen, J.J., *Electromagnetic Interference in CMOS Circuits.*, IEEE Int. Symp. On Electromagnetic Compatibility, 1988, pp 471-472.

[Lomelino & Iyer, 1986]
Lomelino, D., & Iyer, R., *Error Propagation in a Digital Avionic Processor : A Simulation Based Study.*, NASA CR-176501, Univ. of Illinois, 1986.

[Lu, 1980]
Lu, D.J., *Watchdog Processors and VLSI.*, Proc. Nat. Electronics Conf. (Chicago), Vol. 34, 1980, pp 240-245.

[Lu, 1982]
Lu, D.J., *Watchdog Processors and Structural Integrity Checking.*, IEEE Trans. Computers, Vol. 31, No. 7, 1982, pp 240-245.

[Madeira et al, 1990]
Madeira, H., Quadros, G., & Silva, J.G., *Experimental Evaluation of a Set of Simple Error Detection Mechanisms.*, Microprocessing and Microprogramming, Vol. 30, 1990, pp 513-520.

[Mahmood & McCluskey, 1985]
Mahmood, A., & McCluskey, E.J., *Watchdog Processors : Error Coverage and Overhead.*, Proc. 15th Fault Tolerant Computing Symp., 1985, pp 214-219.

[Mahmood & McCluskey, 1988]
Mahmood, A., & McCluskey, E.J., *Concurrent Error Detection Using Watchdog Processors - A Survey.*, IEEE Trans. Computers, Vol. 37, No. 2, 1988, pp 160-174.

[Mahmood et al, 1983]
Mahmood, A., McCluskey, E.J., & Lu, D.J., *Concurrent Fault Detection Using Watchdog Processors and Assertions.*, IEEE Proc. Int. Test. Conf., 1983, pp 622-628.

[Marchal & Courtois, 1982]
Marchal, P. & Courtois, B., *On Detecting the Hardware failures Disrupting Programs in Microprocessors.*, FTCS-12, 1982, pp 249-256.

[McCluskey & Wakerly, 1981]
McCluskey, E.J. & Wakerly, J.F., *A Circuit for Detecting and Analysing Temporary Faults.*, Proc. COMPCON, 1981, pp 317-321.

[McConnel, 1981]
McConnel, S.R., *Analysis and Modelling of Transient Errors in Digital Computers.*, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[McConnel & Siewiorek, 1978]
McConnel, S.R. & Siewiorek, D.P., *C.vmp : The Implementation, Performance, and Reliability of a Fault Tolerant Multiprocessor.*, Interim Report, Carnegie-Mellon University, Computer Science Dept., Pittsburgh, PA 15213, USA., March 1978.

[McGough & Swern, 1981]
McGough, J.G., & Swern, F., *Measurement of Fault Latency in Digital Avionic Mini-Processor.*, NASA, Bendix Corp., Part I (Oct. 1981).

[McGough & Swern, 1983]
McGough, J.G., & Swern, F., *Measurement of Fault Latency in Digital Avionic Mini-Processor.*, NASA, Bendix Corp., Part II (Jan 1983).

[Millman, 1979]
Millman, J., *Microelectronics : Digital and Analogue Circuits and Systems.*, McGraw-Hill Inc., 1979.

[Morganti et al, 1978]
Morganti, M., Coppadoro, G., & Ceru, S., *UDET 7116 - Common Control for PCM Telephone Exchange : Diagnostic Software Design and Availability Evaluation.*, Digest 8th Annual Int. Conf. on Fault Tolerant Computing, 1978, pp 16-23.

[Musa, 1975]
Musa, J.D., *A Theory of Software Reliability and its Application.*, IEEE Trans. Software Engineering, Vol. 1, No. 3, 1975, pp 312-327.

[Musa et al, 1987]
Musa, J.D., Iannion, A., & Okumoto, K., *Software Reliability : Measurement, Prediction, Application.*, McGraw-Hill, 1987.

[Namjoo, 1982]
Namjoo, M. *Techniques for Concurrent Testing of VLSI Processor Operation.*, IEEE Int. Test Conf., 1982, pp 461-468.

[Namjoo, 1983]
Namjoo, M. *CERBERUS-16: An Architecture for a General Purpose Watchdog Processor.*, Proc. FTCS-13, 1983, pp 216-219.

[Namjoo & McCluskey, 1982]
Namjoo, M., & McCluskey, E.J., *Watchdog Processors and Capability Checking.*, Proc. FTCS-12, 1982, pp 245-248.

[O'Connor, 1985]
O'Connor, D.T., *Practical Reliability Engineering (Second Edition).*, John Wiley & Sons, 1985.

[Ornstein et al, 1975]
Ornstein, S.M., Crowther, W.R., Kraley, M.F., Bressler, R.D., Michel, A., & Heart, F.E., *PLURIBUS - A Reliable Multiprocessor.*, AFIPS Int. Computer Conf., 1975, pp 551-559.

[Pearson, 1983]
Pearson, J.C., *Reliability of Small Digital Controllers.*, PhD. Thesis, University of Durham (England), 1983.

[Randall, 1975]
Randall, B., *System Structure for Software Fault Tolerance.*, IEEE Trans. Soft. Eng., Vol. 1., 1975, pp 220-232.

[Rigby & Norris, 1990]
Rigby, P., & Norris, M., *The Software Death Cycle.*, IEE UK IT Conference 1990, University of Southampton, March 1990.

[Russel & Sayers, 1989]
Russel, G. & Sayers, I.L., *Advanced Simulation and Test Methodologies for VLSI Design.*, Van Nostrand Reinhold, 1989.

[Saxena & McCluskey, 1990]
Saxena, N.R., & McCluskey, E.J., *Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums.*, IEEE Trans. Computers, Vol. 39, No. 4, 1990, pp 554-559.

[Schmid et al, 1982]
Schmid, M.E., Trapp, R.L., Davidoff, A.E., & Masson, G.M., *Upset Exposure by Means of Abstraction Verification.*, FTCS-12, 1982, pp 237-244.

[Schuette & Shen, 1986]
Schuette, M.A., & Shen, J.P., *Processor Control Flow Monitoring Using Signatured Instruction Streams.*, IEEE Trans. Computers, Vol. 36, No. 3, 1986, pp 264-276.

[Segall et al, 1988]
Segall, Z., Vrsalovic, D., Siewiorek, D., Yaskin, D., Kownacki, J., Barton, J., Dancy, R., Robinson, A., & Lin, T., *FIAT - Fault Tolerant Based Automated Testing Environment.*, Proc. FTCS-18, 1988, pp 102-107.

[Shen & Schuette, 1983]
Shen, J.P., & Schuette, M.A., *On-Line Monitoring Using Signatured Instruction Streams.*, IEEE Proc. Int. Test. Conf., 1983, pp 275-282.

[Shoji, 1987]
Shoji, M., *CMOS Digital Circuit Technology.*, Prentice-Hall, 1987.

[Siewiorek & Swarz, 1982]
Siewiorek, D.P. & Swarz, R.S., *The Theory and Practice of Reliable Systems.*, Digital Press (Bedford, MA.), 1982.

[Siewiorek et al, 1978a]
Siewiorek, D.P., Kini, V., Masburn, H., McConnel, S., & Tsao, M., *A Case Study of C.mmp, Cm*, and C.vmp : Part 1 - Experiences with Fault Tolerance in Multiprocessor Systems.*, Proc. IEEE, Vol. 66, No. 10, 1978, pp 1178-1199.

[Siewiorek et al, 1978b]
Siewiorek, D.P., Kini, V., Joobbani, R., & Bellis, H., *A Case Study of C.mmp, Cm*, and C.vmp : Part 2 - Predicting and Calibrating Reliability of Microprocessor Systems.*, Proc. IEEE, Vol. 66, No. 10, 1978, pp 1200-1220.

[Sommerville, 1985]
Sommerville, I., *Software Engineering.*, Addison-Wesley Publishing Company, 1985.

[Sosnowski, 1986a]
Sosnowski, J., *Evaluation of Transient Hazards in Microprocessor Controllers.*, Proc. FTCS-16, 1986, pp 364-369.

[Sosnowski, 1986b]
Sosnowski, J., *Transient Fault Effects in Microprocessor Controllers.*, Reliability Technology - Theory and Applications, (eds. Moltoft, J., & Jensen, F.), North-Holland Press, 1986, pp 329-348.

[Stiffler, 1980]
Stiffler, J.I., *Robust Detection of Intermittent Faults.*, IEEE Proc. Int. Symp. on Fault Tolerant Computing, 1980, pp 216-218.

[Tanenbaum, 1990]
Tanenbaum, A.S., *Structured Computer Organization.*, Prentice-Hall International, 1990.

[Taylor et al, 1980]
Taylor, D.J., Morgan, D.E., & Black, J.P., *Redundancy in Data Structures : Improving Software Fault Tolerance.*, IEEE Trans. Software Engineering, Vol. 6, No. 6, 1980, pp 585-594.

[Trachtenberg, 1990]
Trachtenberg, M., *A General Theory of Software Reliability Modelling.*, IEEE Trans. Reliability, Vol. 39, No. 1, 1990, pp 92-96.

[Wilken & Shen, 1987]
Wilken, K.D., & Shen, J.P., *Embedded Signature Monitoring Analysis and Technique.*, IEEE Proc. Int. Test. Conf., 1987, pp 324-333.

[Wingate & Preece, 1991] {*Copy in Appendix E*}
Wingate, G.A.S. & Preece, C., *Analysis of Failure Data Collected From a TMR Microprocessor Controller.*, Microprocessing & Microprogramming, Vol. 32, 1991, pp 861-868.

[Woodbury & Shin, 1990]
Woodbury, M.H., & Shin, K.G., *Measurement and Analysis of Workload Effects on Fault Latency in Real-Time Systems.*, IEEE Trans. Soft. Engineering, Vol. 16, No. 2, 1990, pp 212-216.

[Wynne et al, 1988]
Wynne, R.J., Parkinson, J.S., & Lees, A., *The Design and Implementation of a Control System for a Multi-Feed Gas Network.*, IEE Control Conference (Oxford), 1988, pp 70-75.

[Yang et al, 1985]
Yang, X., York, G., Birmingham, W., & Siewiorek, D., *Fault Recovery of Triplicated Software on the Intel iAPX 432.*, Distributed Computing Systems, 1985, pp 138-143.

[Yau & Chen, 1980]
Yau, S.S., & Chen, F.C., *An Approach to Concurrent Control Flow Checking.*, IEEE Trans. Soft. Eng., Vol. 6, No. 2, 1980, pp 126-137.

Appendix A

INSTRUCTION SET PARAMETERS

APPENDIX A

INSTRUCTION SET PARAMETERS

A.1. Introduction

The model of erroneous microprocessor behaviour described in Chapter 3 is based on the instruction mix of the processor software. The evaluation of the model for a selection of 8, 16, and 32-bit microprocessors, presented in Chapter 4, requires the mix of their respective instruction sets. This appendix contains details of the instruction set parameters used in Chapter 4. The microprocessors considered are: (8-bit) MC 6800, Intel 8048, and Intel 8085 ; (16-bit) Intel 8086, MC 68000, and MC 68010 ; (32-bit) AMD 29000-D, MC 68020, and Intel 80386. The notations MC and AMD specify 'Motorola Corporation' and 'Advanced Micro Devices' respectively.

Data for the 8-bit processors is taken from Halse [1984]. Data parameters for the 16 and 32-bit microprocessors has been evaluated from appropriate manufacturers' manuals listed in the bibliography.

A.2. Instructions Influencing Program Flow

Program flow through software is determined by the content of the microprocessor program counter. The instruction set of a microprocessor contains three types of instruction, each affecting the program counter content in a different way. Firstly, 'non-jump' instructions are classified as those instructions that perform some operation and increment the program counter to the next logical instruction location in the address space. Secondly, 'unconditional jump' instructions specify the program counter to contain the next sequential instruction address which may not be the next logical instruction in memory. Finally, 'conditional jump' instructions test a specified parameter value and, if successful, generate a branch like the unconditional jump instruction. If the test is unsuccessful then the program counter increments to the next logical instruction address like the non-jump instruction.

Microprocessor architectures implementing a ROM opcode decoder generate operations for all possible instruction opcode formats, i.e. 2^n instructions where the opcode has n bits. ROM decoders often have some redundancy, for instance the MC 68000, MC 68010, MC 68020, where $n = 16$ have more ROM opcode values than implemented operations and hence there is a large number of undeclared instructions. The AMD Am29000 is a Reduced Instruction Set Computer (RISC) and implements a ROM decoder for opcodes of 8 bits leaving fewer unused opcode formats (undeclared instructions). Processors not implementing a ROM decoder, directly interpret the opcode through digital circuitry (examples include the Intel 8048, Intel 8085, Intel 8086, and Intel 80386). These microprocessors have all their instructions hardwire defined, although some instructions may appear undeclared because the manufacture withholds information. Undeclared instructions can be non-jumps, conditional jumps, or unconditional jumps, and knowledge of these instructions may be of benefit to the programmer. Tables A.1. to A.9. summarise details of the instruction sets for a variety of processors.

A.3. Microprocessor Jump Type Instruction Data

A fundamental characteristic associated with the program flow during erroneous microprocessor behaviour is the 'erroneous jump'. Jump type instructions can be catagorised in terms of the nature of their branch operation. Chapter 3 defines the following catagories of jump instruction ;

Restart (RT)	:	Leads to a jump to a predefined location in the address space.
Return (RN)	:	Leads to a jump to an address held in a stack.
Stop/Wait (SW)	:	Leads to cessation of processing and requires an interrupt or hardware reset to exit this state.
Unspecified Jump (UJ)	:	Leads to a jump to a new location in the address space, determined by volatile memory content.

Deriving the distribution of restart, return, stop/wait, and unspecified jump instructions in an instruction set requires evaluation of all undeclared instructions.

Within the selection of microprocessors analysed in this appendix, the MC 68000, MC 68010, AMD Am29000-D, MC 68020, and Intel 80386 specify their undeclared instructions to generate a restart. Investigation has shown that the undeclared instructions in the remaining processors can be restart, return, stop/wait, or unspecified jump [Halse, 1984].

The distribution of different jump type instructions for the MC 6800, Intel 8048, Intel 8085, and Itel 8086 microprocessors are shown in Tables A.10 to A.13. respectively. Undeclared instructions with a jump type operation are identified in the Tables by the mnemonic '***'.

The MC 68000, MC 68010, and MC 68020 instruction sets are upwardly compatible; the distribution of their jump type instructions is collated in Table A.14. Similarly, the jump type instruction distributions for the AMD Am29000 and Intel 80386 microprocessors are shown in Table A.15. and Table A.16. respectively.

Table A.1. : MC 6800 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	173	49	222
Cond. Jump	14	1	15
Uncond. Jump	10	9	19
Total	197	59	256

Table A.2. : Intel 8048 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	183	22	205
Cond. Jump	20	3	23
Uncond. Jump	27	1	28
Total	230	26	256

Table A.3. : Intel 8085 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	209	7	216
Cond. Jump	24	3	27
Uncond. Jump	13	0	13
Total	246	10	256

Table A.4. : Intel 8086 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	262	25	287
Cond. Jump	20	0	20
Uncond. Jump	27	6	33
Total	309	31	340

Table A.5. : MC 68000 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	39039	0	39039
Cond. Jump	4224	0	4224
Uncond. Jump	79	22194	22273
Total	43342	22194	65536

Table A.6. : MC 68010 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	39217	0	39217
Cond. Jump	4224	0	4224
Uncond. Jump	80	22015	22273
Total	43521	22015	65536

Table A.7. : AMD 29000 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	151	0	151
Cond. Jump	27	0	27
Uncond. Jump	8	70	78
Total	186	70	256

Table A.8. : MC 68020 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	41623	0	41623
Cond. Jump	4832	0	4832
Uncond. Jump	140	18941	19081
Total	46595	18941	65536

Table A.9. : Intel 80386 Microprocessor

Instr. Type	Declared	Undeclared	Total
Non-Jump	333	0	333
Cond. Jump	37	0	37
Uncond. Jump	21	225	246
Total	391	225	616

Tables A.1-9. Microprocessor Instruction Set Evaluation

Instr. = Instruction ; Cond. = Conditional ; Uncond. = Unconditional

Unconditional Jumps			Conditional Jumps		
Code	Mnemonic	Type	Code	Mnemonic	Type
20	BRA	UJ	21	***	UJ
38	***	RN	22	BHI	UJ
39	RTS	RN	23	BLS	UJ
3A	***	RN	24	BCC	UJ
3B	RTI	RN	25	BCS	UJ
3C	***	SW	26	BNE	UJ
3D	***	SW	27	BEQ	UJ
3E	WAIT	SW	28	BVC	UJ
3F	SWI	RT	29	BVS	UJ
6E	JMP	UJ	2A	BPL	UJ
7E	JMP	UJ	2B	BMI	UJ
8D	BSR	UJ	2C	BGE	UJ
9D	***	SW	2D	BLT	UJ
AD	JSR	UJ	2E	BGT	UJ
BD	JSR	UJ	2F	BLE	UJ
CD	***	UJ			
DD	***	SW			
ED	***	UJ			
FD	***	UJ			

Table A.10. : MC 6800 Jump Instructions

Code = Hexadecimal Instruction Opcode

Unconditional Jumps						Conditional Jumps					
Code	Mnemonic	Type	Code	Mnemonic	Type	Code	Mnemonic	Type	Code	Mnemonic	Type
04	JMP	UJ	E8	DJNZ	UJ	06	***	UJ	D2	JB6	UJ
14	CALL	UJ	E9	DJNZ	UJ	12	JB0	UJ	E6	JNC	UJ
24	JMP	UJ	EA	DJNZ	UJ	16	JTF	UJ	F2	JB7	UJ
34	CALL	UJ	EB	DJNZ	UJ	26	JNT0	UJ	F6	JC	UJ
44	JMP	UJ	EC	DJNZ	UJ	32	JB1	UJ			
54	CALL	UJ	ED	DJNZ	UJ	36	JT0	UJ			
64	JMP	UJ	EE	DJNZ	UJ	46	JNT1	UJ			
74	CALL	UJ	EF	DJNZ	UJ	52	JB2	UJ			
83	RET	RN	F4	CALL	UJ	56	JT1	UJ			
84	JMP	UJ				66	***	UJ			
93	RETR	RN				72	JB3	UJ			
94	CALL	UJ				76	JF1	UJ			
A4	JMP	UJ				86	JN1	UJ			
B3	JMP	UJ				92	JB4	UJ			
B4	CALL	UJ				96	JNZ	UJ			
C4	JMP	UJ				A6	***	UJ			
D4	CALL	UJ				B2	JB5	UJ			
D6	***	UJ				B6	JF0	UJ			
E4	JMP	UJ				C6	JZ	UJ			

Table A.11. : Intel 8048 Jump Instructions

Code = Hexadecimal Instruction Opcode

Unconditional Jumps			Conditional Jumps					
Code	Mnemonic	Type	Code	Mnemonic	Type	Code	Mnemonic	Type
76	HLT	SW	C0	RNZ	RT	F0	RP	RN
C3	JMP	UJ	C2	JNZ	UJ	F2	JP	UJ
C7	RST0	RT	C4	CNZ	UJ	F4	CP	UJ
C9	RET	RN	C8	RZ	RT	F8	RM	RN
CD	CALL	UJ	CA	JZ	UJ	FA	JM	UJ
CF	RST1	RT	CB	***	RT	FC	CM	UJ
D7	RST2	RT	CC	CZ	UJ	FD	***	UJ
DF	RST3	RT	D0	RNC	RT			
E7	RST4	RT	D2	JNC	UJ			
E9	PCHL	UJ	D4	CNC	UJ			
EF	RST5	RT	D8	RC	RT			
F7	RST6	RT	DA	JC	UJ			
FF	RST7	RT	DC	CC	UJ			
			DD	***	UJ			
			E0	RP0	RN			
			E2	JP0	UJ			
			E4	CP0	UJ			
			E8	RPE	RN			
			EA	JPE	UJ			
			EC	CPE	UJ			

Table A.12. : Intel 8085 Jump Instructions

Code = Hexadecimal Instruction Opcode

Unconditional Jumps						Conditional Jumps		
Code	Mnemonic	Type	Code	Mnemonic	Type	Code	Mnemonic	Type
0F	***	RN	EB	JMP	UJ	70	JO	UJ
2E	SEG	UJ	F0	LOCK	SW	71	JNO	UJ
3E	SEG	UJ	F1	***	UJ	72	JB	UJ
9A	CALL	UJ	F2	REP	UJ	73	JNB	UJ
9B	WAIT	UJ	F4	HALT	SW	74	JE	UJ
C0	***	RN	FE[xx01][0xxx]	CALL	UJ	75	JNE	UJ
C1	***	RN	FE[xx01][1xxx]	CALL	UJ	76	JBE	UJ
C2	RET	RN	FE[xx10][0xxx]	JMP	UJ	77	JNBE	UJ
C3	RET	RN	FE[xx10][1xxx]	JMP	UJ	78	JS	UJ
C8	***	UJ	FF[xx01][0xxx]	CALL	UJ	79	JNS	UJ
C9	***	RN	FF[xx01][1xxx]	CALL	UJ	7A	JP	UJ
CA	RET	RN	FF[xx10][0xxx]	JMP	UJ	7B	JNP	UJ
CB	RET	RN	FF[xx10][1xxx]	JMP	UJ	7C	JL	UJ
CC	INT	RT				7D	JNL	UJ
CD	INT	RT				7E	JLE	UJ
CF	IRET	RN				7F	JNLE	UJ
E2	LOOP	UJ				E0	LOOPNZ	UJ
E8	CALL	UJ				E1	LOOPZ	UJ
E9	JMP	UJ				E3	JCXZ	UJ
EA	JMP	UJ				F3	REPZ	UJ

Table A.13. : Intel 8086 Jump Instructions

Code = Hexadecimal Instruction Opcode ; [ ] = Hexadecimal Digit ;
x = Further Specification Required

Unconditional Jumps				Conditional Jumps							
Code	Quantity	Mnemonic	Type	Code	Quantity	Mnemonic	Type	Code	Quantity	Mnemonic	Type
4AFC	1	ILLEGAL	RT	50C[1xxx]	8	DBT	UJ	63xx	256	BLS	UJ
4E4x	16	TRAP	RT	51C[1xxx]	8	DBF	UJ	64xx	256	BCC	UJ
4E70	1	RESET	RT	52C[1xxx]	8	DBHI	UJ	65xx	256	BCS	UJ
4E72	1	STOP	SW	53C[1xxx]	8	DBLS	UJ	66xx	256	BNE	UJ
4E73	1	RTE	RN	54C[1xxx]	8	DBCC	UJ	67xx	256	BEQ	UJ
4E75	1	RTS	RN	55C[1xxx]	8	DBCS	UJ	68xx	256	BVC	UJ
4E76	1	TRAPV	RT	56C[1xxx]	8	DBNE	UJ	69xx	256	BVS	UJ
4E77	1	RTR	RN	57C[1xxx]	8	DBEQ	UJ	6Axx	256	BPL	UJ
4E[10xx]x	28	JSR	UJ	58C[1xxx]	8	DBVC	UJ	6Bxx	256	BMI	UJ
4E[11xx]x	28	JMP	UJ	59C[1xxx]	8	DBVS	UJ	6Cxx	256	BGE	UJ
				5AC[1xxx]	8	DBPL	UJ	6Dxx	256	BLT	UJ
MC 68010				5BC[1xxx]	8	DBMI	UJ	6Exx	256	BGT	UJ
4E74	1	RTD	RN	5CC[1xxx]	8	DBGE	UJ	6Fxx	256	BLE	UJ
				5DC[1xxx]	8	DBLT	UJ				
MC 68020				5EC[1xxx]	8	DBGT	UJ	MC 68020			
06[1xxx]x	52	CALLM	RT	5FC[1xxx]	8	DBLE	UJ	5xF[1xxx]	48	TRAPcc	RT
484[1xxx]	8	BKPT	UJ	60xx	256	BRA	UJ	F[xxx0][1xxx]x	448	cpBcc	UJ
				61xx	256	BSR	UJ	F[xxx0]4[1xxx]	56	cpDBcc	UJ
				62xx	256	BHI	UJ	F[xxx0]7[1xxx]	56	cpTRAPcc	RT

Table A.14. : MC 68000, MC 68010, and MC68020 Jump Instructions

Code = Hexadecimal Instruction Opcode ; [ ] = Hexadecimal Digit ; x = Further Specification Required

Unconditional Jumps			Conditional Jumps					
Code	Mnemonic	Type	Code	Mnemonic	Type	Code	Mnemonic	Type
88	IRETINV	RN	50	ASLT	RT	A5	JMPF	UJ
89	HALT	SW	51	ASLT	RT	AC	JMPT	UJ
8C	IRET	RN	52	ASLTU	RT	AD	JMPT	UJ
A0	JMP	UJ	53	ASLTU	RT	B4	JMPFDEC	UJ
A1	JMP	UJ	54	ASLE	RT	B5	JMPFDEC	UJ
A8	CALL	UJ	55	ASLE	RT	CC	JMPTI	UJ
A9	CALL	UJ	56	ASLEU	RT			
C8	CALLI	UJ	57	ASLEU	RT			
			58	ASGT	RT			
			59	ASGT	RT			
			5A	ASGTU	RT			
			5B	ASGTU	RT			
			5C	ASGE	RT			
			5D	ASGE	RT			
			5E	ASGEU	RT			
			5F	ASGEU	RT			
			70	ASEQ	RT			
			71	ASEQ	RT			
			72	ASNEQ	RT			
			73	ASNEQ	RT			
			A4	JMPF	UJ			

Table A.15. : AMD 29000 Jump Instructions

Code = Hexadecimal Instruction Opcode

Unconditional Jumps			Conditional Jumps					
Code	Mnemonic	Type	Code	Mnemonic	Type	Code	Mnemonic	Type
2E	SEG	UJ	0F00	JO	UJ	75	JNZ	UJ
3E	SEG	UJ	0F01	JNO	UJ	76	JBE	UJ
9A	CALL	UJ	0F02	JB	UJ	77	JNBE	UJ
9B	WAIT	SW	0F03	JNB	UJ	78	JS	UJ
C2	RET	RN	0F04	JZ	UJ	79	JNS	UJ
C3	RET	RN	0F05	JNZ	UJ	7A	JP	UJ
CA	RET	RN	0F06	JBE	UJ	7B	JNP	UJ
CB	RET	RN	0F07	JNBE	UJ	7C	JL	UJ
CC	INT	RT	0F08	JS	UJ	7D	JNL	UJ
CD	INT	RT	0F09	JNS	UJ	7E	JLE	UJ
CE	INTO	RT	0F0A	JP	UJ	7F	JNLE	UJ
CF	IRET	RN	0F0B	JNP	UJ	E0	LOOPE	UJ
E2	LOOP	UJ	0F0C	JL	UJ	E1	LOOPNE	UJ
E8	CALL	UJ	0F0D	JNL	UJ	E3	JCXZ	UJ
E9	JMP	UJ	0F0E	JLE	UJ	F2	REPNE	UJ
EA	JMP	UJ	0F0F	JNLE	UJ	F3	REPE	UJ
EB	JMP	UJ	70	JO	UJ			
FF[xx01][0xxx]	CALL	UJ	71	JNO	UJ			
FF[xx01][1xxx]	CALL	UJ	72	JB	UJ			
FF[xx10][0xxx]	JMP	UJ	73	JNB	UJ			
FF[xx10][1xxx]	JMP	UJ	74	JZ	UJ			

Table A.16. : Intel 80386 Jump Instructions

Code = Hexadecimal Instruction Opcode ; [ ] = Hexadecimal Digit ;
x = Further Specification Required

Appendix B

THE DESIGN OF AN ACCESS GUARDIAN

APPENDIX B

THE DESIGN OF AN ACCESS GUARDIAN

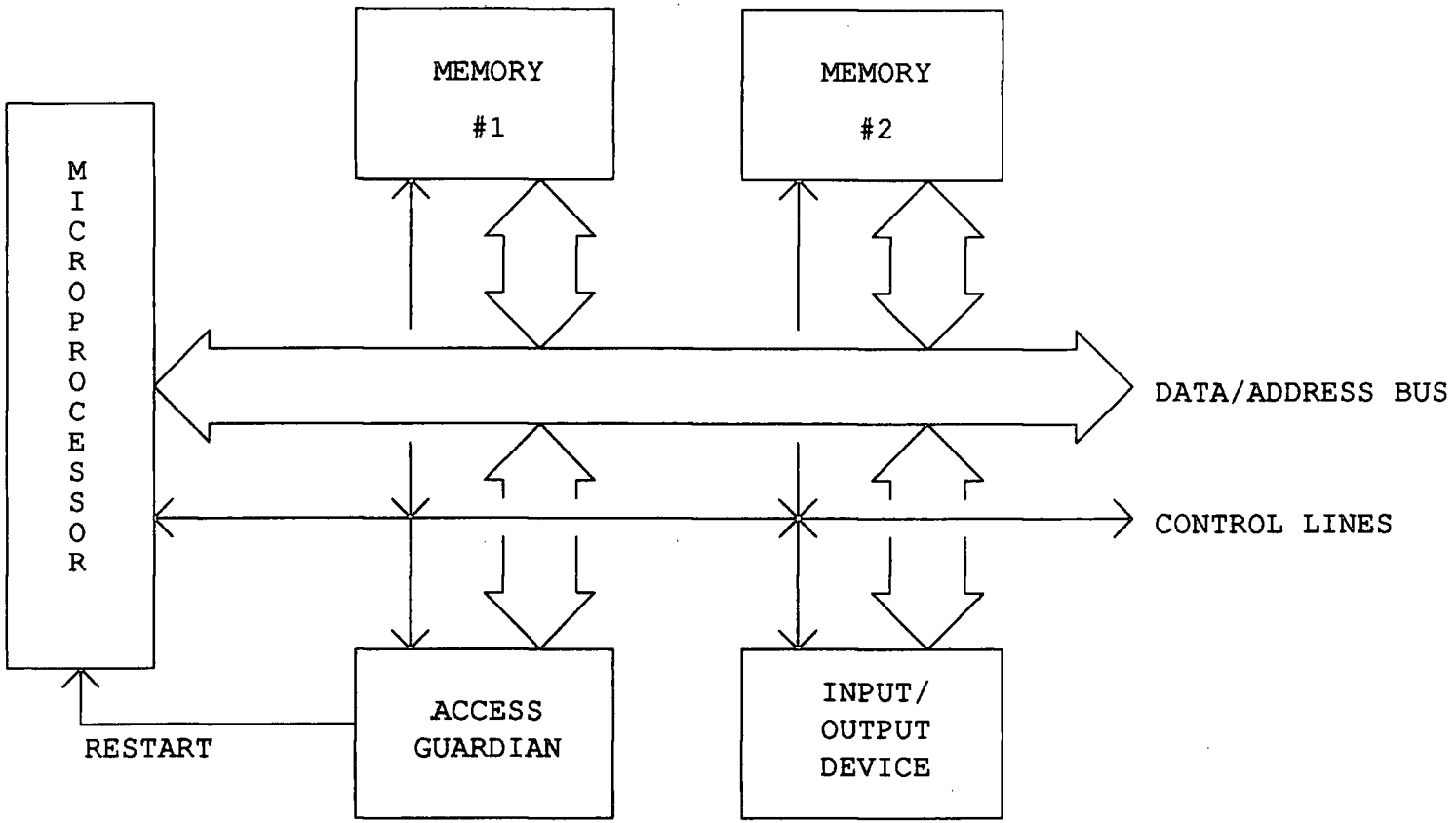
B.1. Introduction

This appendix describes the design of an Access Guardian proposed in Chapter 5. The Access Guardian detects whether or not invalid address lines are activated by the microprocessor whose operation it is montioring, and if so, impresses an interrupt signal to the microprocessor. The design is validated through a gate-level simulation, and the hardware requirement is listed. The topology of a microprocessor system incorporating an Access Guardian is shown in Figure B.1.

B.2. An Access Guardian Design

The Access Guardian design presented here monitors a dedicated address bus for access outside a contiguous 16 MByte block of memory, and has a required interrupt latency of ten clock cycles. Whilst particular microprocessor applications are expected to have a more complex Access Guardian specification, the requirements used here are sufficient to indicate design implications.

The Access Guardian design is based on the interaction of three functional units: the 'address decoder', the 'restart generator', and the 'timer unit'. The general function of the Access Guardian is shown in Figure B.2. The 'address decoder' generates a signal when invalid address lines are activated. This signal is then processed with 'timer unit' status information by the 'restart generator' to produce an interrupt signal for the application processor. The interrupt signal must exist slightly in excess of the microprocessor interrupt latency. The interrupt latency is the length of time an interrupt must exist to guarantee processing by the microprocessor. Assuming the interrupt is given highest priority the processor will detect it following the execution of the present instruction. The interrupt signal must therefore be just longer than the longest execution time required by any instruction.

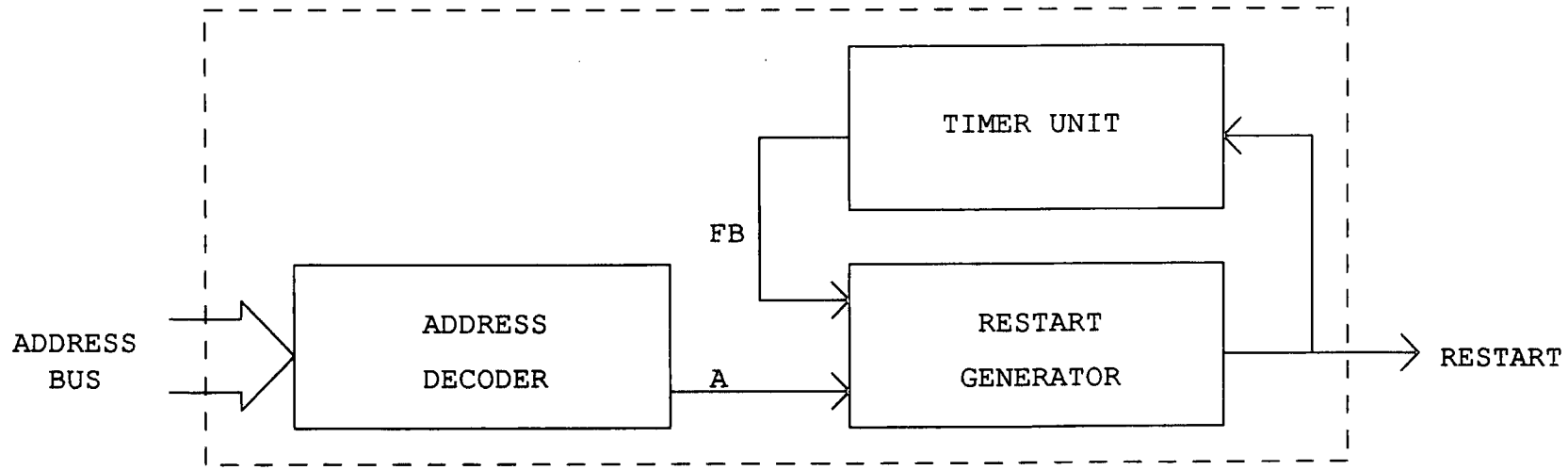


This diagram shows the typical topology of a microprocessor based controller incorporating an Access Guardian.

RESTART is the interrupt signal impressed by the Access Guardian upon the microprocessor to notify detection of invalid address line activity indicative of erroneous execution.

Figure B.1. : Embedded Access Guardian

EMBEDDED ACCESS GUARDIAN		
Size A	Document Number Guy Wingate ©A	REV
Date: May 15, 1990	Sheet	of



This diagram shows the functional units of the Access Guardian. Not all address lines may be required by the address decoder. Signal A denotes detection of invalid address line activity. RESTART is the interrupt signal impressed upon the microprocessor which indicates erroneous execution. The timer unit holds the generated RESTART signal for a duration that guarantees processor receipt. The feedback signal FB resets the RESTART at the end of the timed period if there is no further invalid address line activity. Continuing invalid address line activity initiates a further duration of the RESTART signal output by the restart generator.

The address decoder, restart generator, and timer unit all take the system input signals CLK (clock) and MRESET (manual reset).

Figure B.2. : Access Guardian

ACCESS GUARDIAN		
Size A	Document Number Guy Wingate ©B	REV
Date: May 15, 1990	Sheet	of

B.3. The Address Decoder

The 'address decoder' determines whether or not invalid address lines have been activated. An example decoder is shown in Figure B.3. for a Motorola 68000 address bus (address lines 'A0','A1',...'A23' specifying a 16 MByte address space) where only the least significant 16 MByte of memory is used. A simple OR function for the four most significant address lines ('A23','A22','A21', and 'A20') determines an invalid access and generates a signal 'A'. The 'address decoder', however, will be more complex if sections of the used area are dispersed across the address space, or if the address bus is multiplexed with the data bus as in the Intel 8086 microprocessor.

B.4. The Restart Generator

The 'restart generator' consists of control logic driving a Set-Reset flip-flop (SRFF). The control circuitry determines the logic values for the SRFF depending on the Access Guardian operating conditions. The controller takes the input 'A' (from the address decoder), the manual reset line ('MRESET'), and the feedback signal 'FB' (from the timer). The SRFF has inputs 'S' and 'R', and output 'Q'. Table B.1. shows the truth table for the control logic to drive the SRFF. The following expressions for 'S' and 'R' are developed from the truth table.

$$S = A.\overline{MRESET} \qquad (B.1.)$$

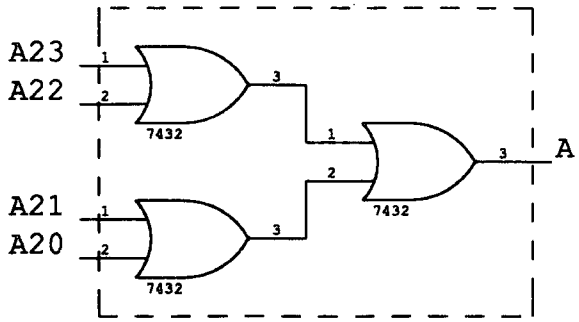
$$R = MRESET \; + \; FB.\overline{A}.\overline{MRESET} \qquad (B.2.)$$

Applying DeMorgan's Theorem yields to equation (B.1.),

$$S = \overline{\overline{\overline{A}.\overline{MRESET}}} \qquad (B.3.)$$

$$S = \overline{\overline{A} \; + \; MRESET} \qquad (B.4.)$$

and to equation (B.2.),



This is a simple example of an address decoder. The address decoder determines whether or not the microprocessor has exercised one of the address bus lines A23, A22, A21, or A20. The complexity of the address decoder is dependent upon the memory configuration employed by the microprocessor application, and whether or not the microprocessor has a multiplexed data/address bus.

A is the signal (active high) that denotes invalid address line activation.

Figure B.3. : Address Decoder

ADDRESS DECODER		
Size A	Document Number Guy Wingate #1	REV
Date: May 14, 1990	Sheet	of

A	MRESET	FB	S	R
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

Table B.1. : Truth Table for SRFF Control Logic in Restart Generator.

S_n	R_n	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	?

Table B.2. : Set-Reset Flip Flop Transition Table

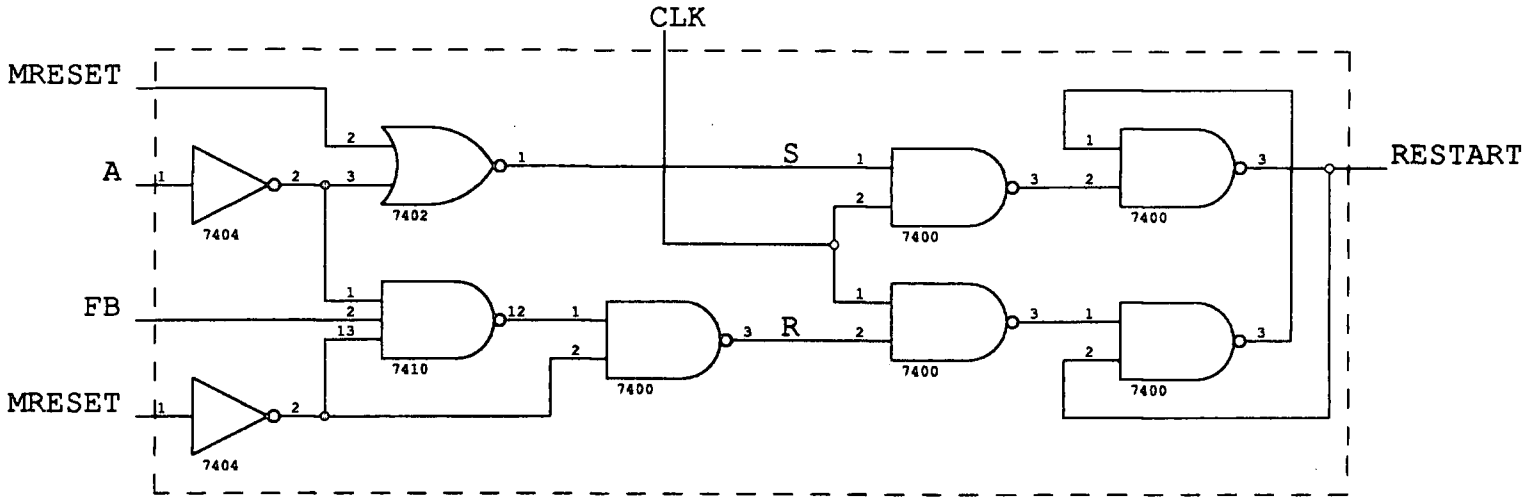
$$R = \overline{\overline{MRESET \ + \ FB.\overline{A}.MRESET}} \qquad (B.5.)$$

$$R = \overline{\overline{MRESET.FB.\overline{A}.MRESET}} \qquad (B.6.)$$

The SRFF is enabled by an input clock signal ('CLK'), and its output 'Q' is dependent upon the input control signals 'S' and 'R'. The relationship between 'Q', 'S', and 'R' is shown in the transition table, see Table B.2. The output 'Q' is set high when an invalid address is decoded and there is no manual reset and no feedback signal indicating the continuing activity of a previously identified invalid address line activity. The output 'Q' remains at the same logic value after being set. The SRFF control circuitry resets the output 'Q' to a low when either the manual reset is exerted, or the feedback signals completed processing of the Access Guardian, and there is no current address line discrepancy detected by the address decoder. The logic design for the 'restart generator' implementing equations (B.3.) and (B.5.) to drive the SRFF is shown in Figure B.4.

B.5. The Timer Unit

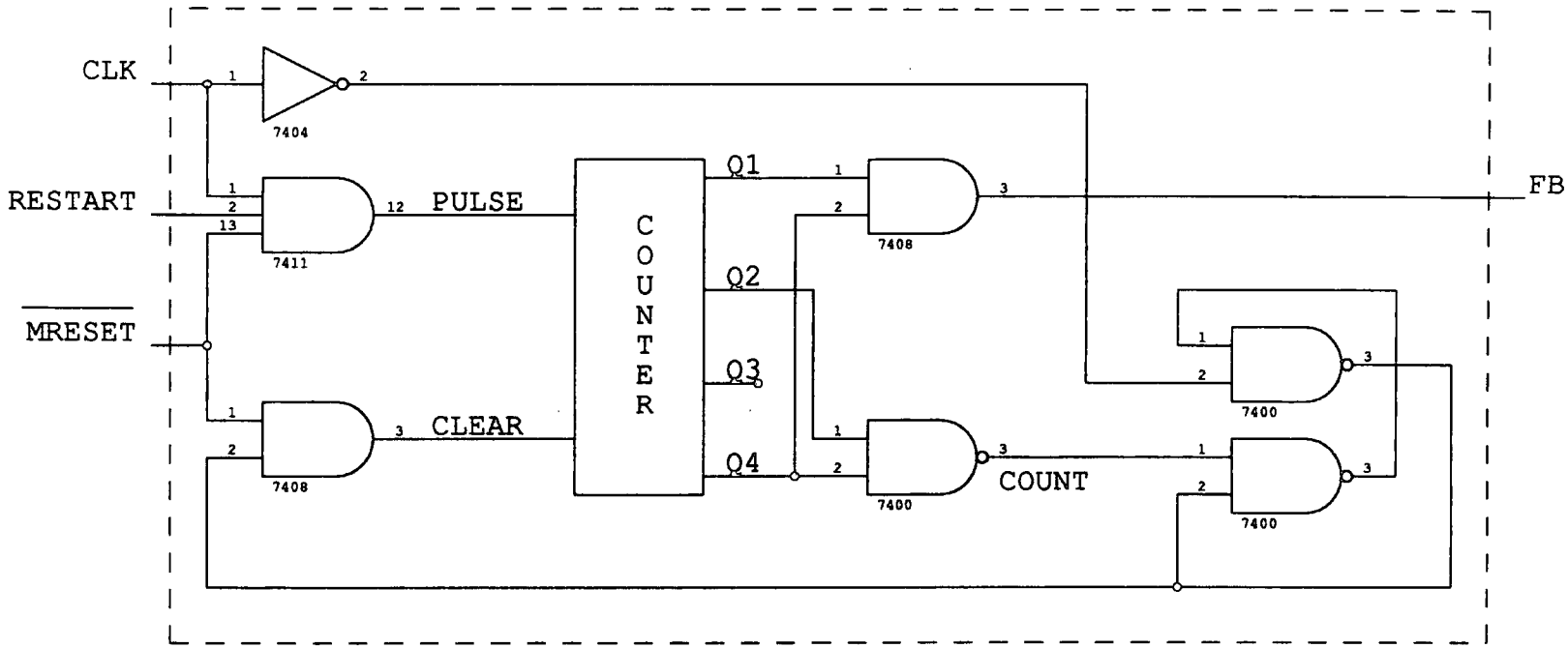
A 'timer unit' is used to hold the interrupt signal for the required interrupt latency period and is shown in Figure B.5. This unit takes as inputs the RESTART interrupt signal, the manual reset signal ('MRESET'), and the clock signal ('CLK'). The 'timer unit' generates a feedback signal ('FB') which is used by the 'restart generator'. Initially the clock, restart interrupt, and inverse manual reset signal are put through a logic AND gate to produce a control signal 'PULSE'. The 'PULSE' line is used to drive, together with the clock signal 'CLK', the ripple counter. The ripple counter consists of a series of master-slave JK flip-flops active on the negative edge of the clock signal ('CLK'). The 'PULSE' signal provides the clocking signal to the first master-slave JK flip-flop to generate an output 'Q1'. The output 'Q1' is used as the clocking signal for the second master-slave JK flip-flop. This method of connecting the master-slave JK flip-flops is repeated throughout the ripple counter.



MRESET is the manual reset signal
A is the output signal from the address decoder
FB is the feedback signal from the timer unit
CLK is the system clock signal
S is the 'set' input to the SR flip flop
R is the 'reset' input to the SR flip flop
RESTART is the interrupt signal to be impressed upon the microprocessor

Figure B.4. : Restart Generator

RESTART GENERATOR		
Size A	Document Number Guy Wingate ø2	REV
Date: May 14, 1990	Sheet	of



This example timer unit incorporates a base 10 counter

MRESET is the manual reset signal
CLK is the system clock signal
RESTART is the interrupt signal to be impressed upon the microprocessor
PULSE is the clocking signal for the counter
The COUNTER is negative-edge triggered
COUNT is active high on counter reset
CLEAR resets the counter on either maual reset (MRESET) or
on the positive-edge on the COUNT signal
FB is the feedback signal; set high after counter timeout

Figure B.5. : Timer Unit

TIMER_UNIT		
Size A	Document Number Guy Wingate ◊3	REV
Date: May 15, 1990	Sheet	of

The 'J' and 'K' inputs to the flip-flops are set high. A 'CLEAR' signal is used to reset the 'Qn' outputs of the flip-flops to logic 0. The ripple counter architecture is shown in Figure B.6. [Millman, 1979].

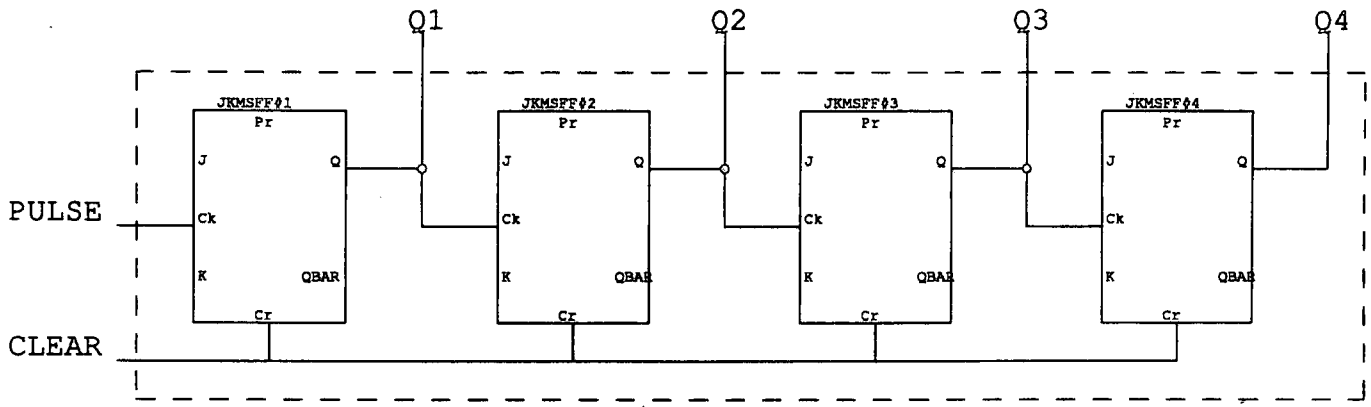
The ripple counter is set to count to a specific number 'C_n' by taking the flip-flop outputs 'Q1', 'Q2',.. 'Qn' and applying them to a NAND logic gate as required. The example ripple counter is a base 10 counter, the NAND gate taking the binary inputs representing decimal 10, 'Q2' and 'Q4'. The NAND gate generates the 'COUNT' signal which indicates a necessary reset of the ripple counter. This signal line could be connected directly to the 'CLEAR' line but there may be timing difficulties due to the unequal internal delays within the ripple counter flip-flops. These timing difficulties are removed by inserting a latch between the 'COUNT' and 'CLEAR' lines.

The latch unit takes the additional input of the clock signal ('CLK') and the manual reset line ('MRESET'). The latch is now reset by the positive edge of the 'COUNT' signal to set the ripple counter low. The ripple counter itself is clocked on the negative edge of the 'PULSE' signal. There are now no timing difficulties. The manual reset line ('MRESET') is used to set the ripple counter outputs low.

The 'Q1', 'Q2',... 'Qn' outputs of the ripple counter are taken to a feedback unit which consists of an AND logic gate. The outputs of 'Qn' represent the binary of (C_n − 1), where the ripple counter is base C_n. In the example, the ripple counter outputs representing decimal 9 (Q1 and Q4) are used, see Figure B.6. The ripple counter can, however, be extended as required to produce the interrupt signal 'RESTART' of necessary duration depending on the microprocessor interrupt latency. The feedback unit generates an automatic reset signal ('FB') to the 'restart generator'.

B.6. Design Simulation

The gate-level design for the Access Guardian was simulated using a CLASSIC (a trademark of Plessey plc.) Gate Array Simulator. The digital circuit description used as input to the simulator, and describing the design presented in this appendix, is shown in Figure B.7. The output of the simulator is shown in Figure B.8. The design of the Access Guardian is shown to operate correctly.



This example shows the general construction for the counter.
It consists of a series of connected Master-Slave JK Flip Flops.
All the flip flops have J,.K, and Pr (preset) inputs tied high.
The flip flop QBAR output is not used. The first flip flop
is clocked by the PULSE signal at its Cr input. Successive
flip flops are clocked by the Q output of the previous flip flop.
The counter is binary, the least significant bit being the output Q1.
Counting occurs on the negative-edge of the PULSE signal due to
the operation of the master and slave internal to each flip flop.
The counter can easily be extended by adding extra Master-Slave
JK Flip Flops onto the counter logic structure.

This logic circuit and its inputs and outputs are defined in
the logic diagram for the timer unit.
JKMSFFØ1, Ø2, Ø3, Ø4 are JK Master-Slave Flip Flops.

Figure B.6. : Ripple Counter Unit

COUNTER UNIT		
Size A	Document Number Guy Wingate Ø4	REV
Date: May 15, 1990	Sheet	of

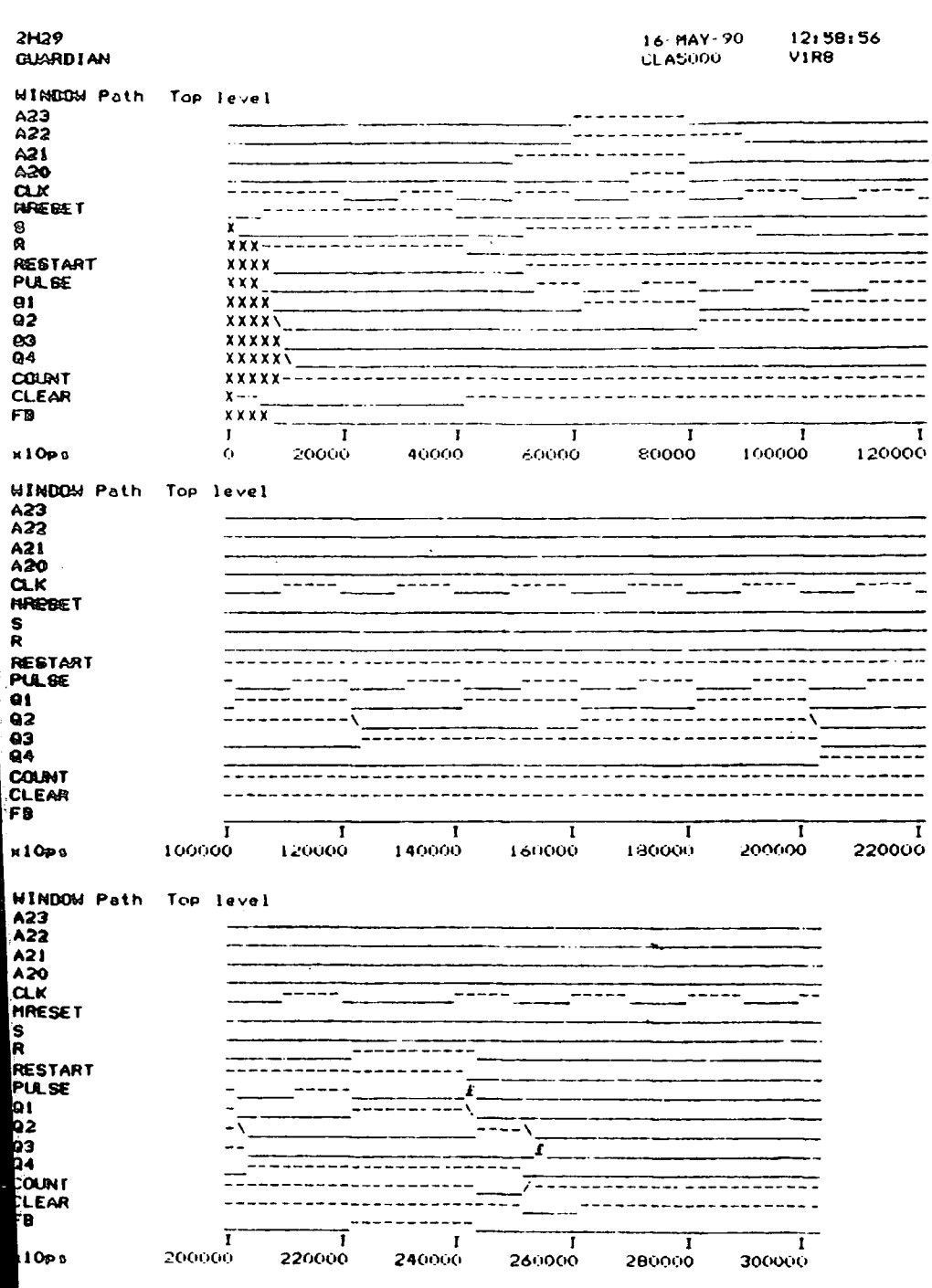


Figure B.7. : Access Guardian Timing Simulation

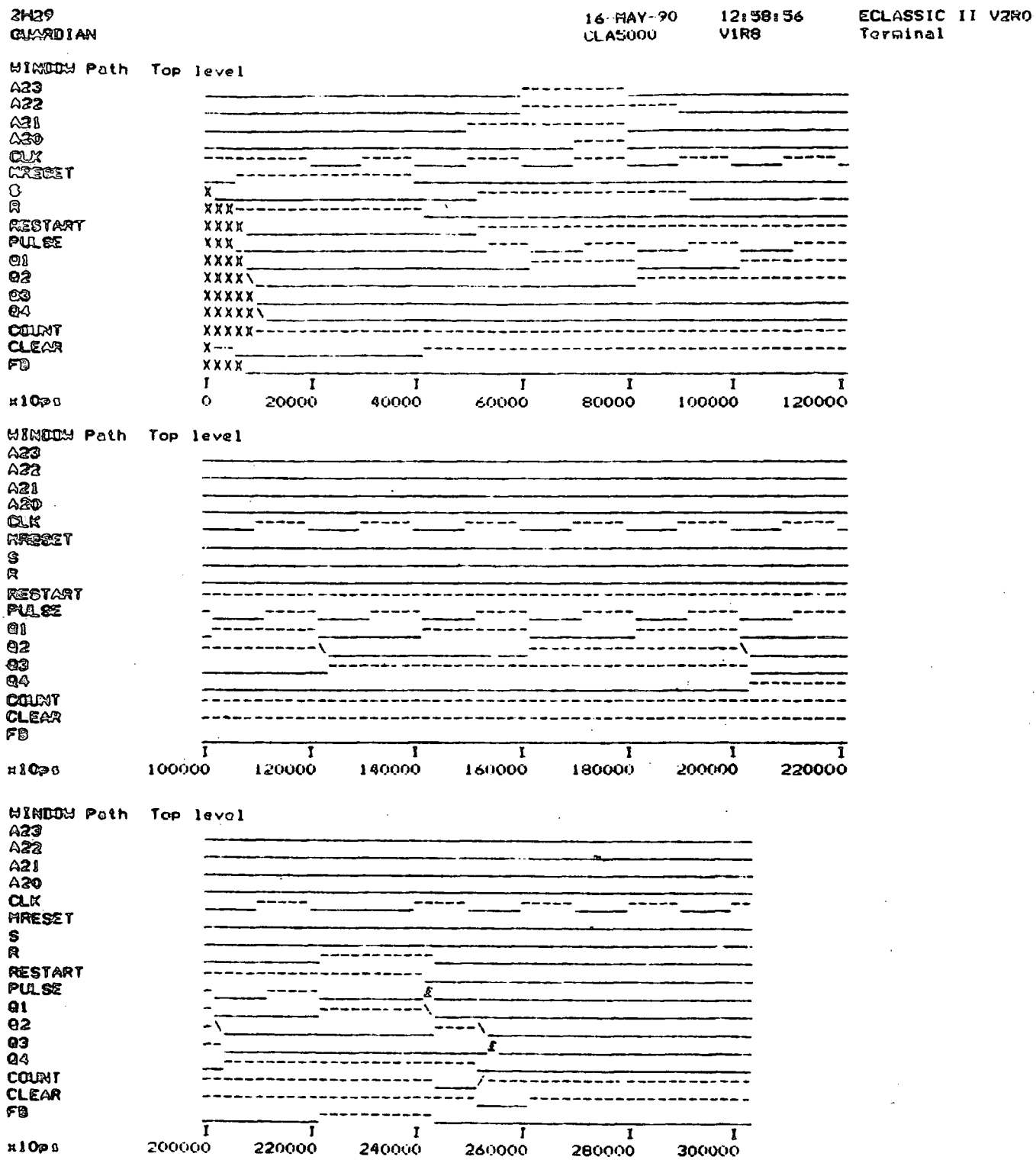


Figure B.7. : Access Guardian Timing Simulation

```
UNITS = 10 pS
SUMM
ARRAYSIZE = 640

CIRCUIT
GUARDIAN[RESTART:A23,A22,A21,A20,CLK,MRESET;]
SUPPLIES/+VDD, -VSS

DECODER1:NOR4 [D1,A23,A22,A21,A20]
DECODER2:2INV [A,D2,D1,A]

RESETSR1:NOR2 [S,D2,MRESET]
RESETSR2:NAND3 [E1,,FB,D2,RESETBAR,-]
RESETSR3:NAND2 [R,RESETBAR,E1]

SR1FF1:NAND2 [S1,S,CLK]
SR1FF2:NAND2 [R1,R,CLK]
SR1FF3:2INV [S2,R2,S1,R1]
SR1FF4:NOR2 [SR1QBAR,S2,Q]
SR1FF5:NOR2 [Q,R2,SR1QBAR]

GATE2:NAND3 [PBAR1,PULSE,Q,CLK,RESETBAR,PBAR1]

JK1FF1:NAND3 [J11,,+,PULSE,JK1QBAR,-]
JK1FF2:NAND3 [K11,,+,PULSE,Q1,-]
JK1FF3:NAND2 [J12,J11,K12]
JK1FF4:NAND3 [K12,,K11,J12,CLEAR,-]
JK1FF5:NAND2 [J13,J12,PBAR1]
JK1FF6:NAND2 [K13,K12,PBAR1]
JK1FF7:NAND2 [Q1,J13,JK1QBAR]
JK1FF8:NAND2 [JK1QBAR,K13,Q1]

JK2FF1:NAND3 [J21,,+,Q1,JK2QBAR,-]
JK2FF2:NAND3 [K21,,+,Q1,Q2,-]
JK2FF3:NAND2 [J22,J21,K22]
JK2FF4:NAND3 [K22,PBAR2,K21,J22,CLEAR,Q1]
JK2FF5:NAND2 [J23,J22,PBAR2]
JK2FF6:NAND2 [K23,K22,PBAR2]
JK2FF7:NAND2 [Q2,J23,JK2QBAR]
JK2FF8:NAND2 [JK2QBAR,K23,Q2]

JK3FF1:NAND3 [J31,,+,Q2,JK3QBAR,-]
JK3FF2:NAND3 [K31,,+,Q2,Q3,-]
JK3FF3:NAND2 [J32,J31,K32]
JK3FF4:NAND3 [K32,PBAR3,K31,J32,CLEAR,Q2]
JK3FF5:NAND2 [J33,J32,PBAR3]
JK3FF6:NAND2 [K33,K32,PBAR3]
JK3FF7:NAND2 [Q3,J33,JK3QBAR]
JK3FF8:NAND2 [JK3QBAR,K33,Q3]

JK4FF1:NAND3 [J41,,+,Q3,JK4QBAR,-]
JK4FF2:NAND3 [K41,,+,Q3,Q4,-]
JK4FF3:NAND2 [J42,J41,K42]
JK4FF4:NAND3 [K42,PBAR4,K41,J42,CLEAR,Q3]
JK4FF5:NAND2 [J43,J42,PBAR4]
JK4FF6:NAND2 [K43,K42,PBAR4]
JK4FF7:NAND2 [Q4,J43,JK4QBAR]
JK4FF8:NAND2 [JK4QBAR,K43,Q4]

COUNTER:NAND2 [COUNT,Q2,Q4]

LATCH1:NAND2 [P,COUNT,RST]
LATCH2:NAND2 [RST,P,CLKBAR]
LATCH3:2INV [CLKBAR,,CLK,-]

RST1:NAND2 [CLEARBAR,RST,RESETBAR]
RST2:2INV [CLEAR,RESETBAR,CLEARBAR,MRESET]

FEEDBACK1:NAND2 [FB1,Q1,Q4]
FEEDBACK2:2INV [FB,,FB1,-]

RESULT:2INV [RESTART,QN,QN,Q]

END
```

Figure B.8. : Access Guardian Circuit Description

B.7. The Design's Hardware Requirement

The hardware requirement for the Access Guardian design is shown in Table B.3. The design specification requires only 60 logic gates, which can be implemented by 17 standard TTL IC parts.

B.8. Summary

The Access Guardian proposed in Chapter 5 has been designed and its operation verified through a gate-level simulation. The design implemented is simple, other designs may be more appropriate in particular applications. The design chosen here operates independently of the processor whose bus activity it is monitoring.

The hardware requirement of the Access Guardian design presented here for a contiguous 16 MByte of used memory, dedicated address bus, and ten cycle interrupt latency, can increase with more complex microprocessor systems. The complexity of the address decoder will increase with a non-contiguous used area of memory and multiplexed busses. The timer unit may be slightly more complicated with particular binary time representations. The restart generator has a fixed logic gate requirement.

Design Specification		TTL Parts Specification		
Gate Description	Number Required	TTL Part	Description	Number Required
2-Input NAND	28	74F00	Quad 2-Input NAND Gate	7
2-Input NOR	3	74F02	Quad 2-Input NOR Gate	1
2-Input Inverter	14	74F04	Hex Inverter	3
3-Input NAND	14	74F10	Triple 3-Input NAND Gate	5
4-Input NOR	1	74F260	Dual 5-Input NOR Gate	1
Total Required	60	Total Required		17

Table B.3. : Access Guardian Parts List.

Appendix C

PARUT AND OTHER RELATED CODE LISTINGS

APPENDIX C

PARUT AND OTHER RELATED CODE LISTINGS

C.1. Introduction

This appendix is designed to be read in conjunction with Chapter 6 which describes the Post-programming Automated Recovery Utility (PARUT). The appendix holds enclosures of the PARUT program listing, and copies of typical input and output files processed by PARUT during its operation. These files, referred to as MICRO_FILE, CODE_FILE, RESULT_FILE, ANALYSIS_FILE, and TRACE_FILE within Chapter 6, are presented respectively with covering notes.

C.2. PARUT Listing

The first enclosure in this appendix is that of the PARUT program. The utility code is written in Pascal. The listing is annotated to aid comprehension of program activity. Table C.1. details, in chronological order, the utility functions and procedures within the listing and is designed for use as an index when examining the PARUT program.

Function or Procedure Name	Called Functions or Procedures
placement_deadlock	None
data_analysis	None
analysis	placement_deadlock
jump_direction	None
interval	None
test_seed	None
seed_details	None
place_seed	None
analysis	jump_direction
	interval
	test_seed
	seed_details
	place_seed
digit	None
build_adb	digit
build_rng	None
build_dis	None
set_jump	None
align_address	None
instru_set	None
build_jumps	instru_set
	set_jump
	align_address
copy_list	None
address_list	None

Function or Procedure Name	Called Functions or Procedures
seed_placement	seed_location
	address_list
	build_jumps
	seed_placement
	build_adb
	copy_list
seed_list	seed_placement
relocate_list	None
stream_list	None
standard_offset	None
align_list	standard_offset
tidy_list	None
print_list	None
main	build_rng
	build_adb
	copy_list
	build_jumps
	seed_list
	relocate_list
	stream_list
	address_list
	align_list
	tidy_list
	print_list
	data_analysis
	analysis

Table C.1. : Chronological Order of Functions in PARUT Listing

```
{*************************************************************************
 *************************************************************************
 **                                                                     **
 **                  AUTOMATED SEED PLACEMENT (PARUT2)                  **
 **                                                                     **
 **                Version 6.0 : 02 Feb 1990 : G Wingate                **
 **                                                                     **
 *************************************************************************
 *************************************************************************}

program asp( input, output, trace_file, analysis_file, code_file, result_file );

CONST                          { Constants for Motorola 68000 Implementation }
  FORWARD = TRUE;              { DEFAULT                                     }
  BACKWARD = FALSE;            { DEFAULT                                     }
  OPCODE = true;               { DEFAULT                                     }
  OPERAND = false;             { DEFAULT                                     }
  NO_SEED = false;             { DEFAULT                                     }
  SEED = 24577;                { BRA Odd Address : 0x6001                    }
  BRA = 24576;                 { BRA 0 Byte Offset : 0x6000                  }
  MAX = 65536;                 { Maximum Byte Size of Address Space          }
  START_ADDRESS = 22000;       { Arbitary Used Area Base Address             }
  HEADER_LENGTH = 6;           { Vitesse 5 'adb' file header length (lines)  }
  INDENT = 9;

{*************************************************************************

DATA_WORD is a record which contains all the information required to
place seeds into a piece of code.

 *************************************************************************}


TYPE
  link = ^data_word;                        { pointer to data_word          }
  data_word = RECORD
                jump_too : link;            { pointer to branch address     }
                jump_from : link;           { pointer to address of branch  }
                next_address : link;        { pointer to next address       }
                last_address : link;        { pointer to last address       }
                address : integer;          { present address               }
                optype : boolean;           { opcode or operand             }
                op : integer;               { address contents              }
                seeded : boolean;           { has jump been seeded          }
                offset : integer;           { address offset here           }
                jump_address : integer;     { location jumped too next      }
                jump_type : integer;        { type of control flow instr    }
                                            { 1  = BCC/BRA/BSR              }
                                            { 2  = CHK                      }
                                            { 3  = DIVS/DIVU                }
                                            { 4  = ILLEGAL                  }
                                            { 5  = JMP                      }
                                            { 6  = JSR                      }
                                            { 7  = RESET                    }
                                            { 8  = RTE                      }
                                            { 9  = RTR                      }
                                            { 10 = RTS                      }
                                            { 11 = STOP                     }
                                            { 12 = TRAP                     }
                                            { 13 = UNDEFINED                }
                end;

  titletype = packed array [ 1..30 ] of char;
  restype = packed array [ 1..6 ] of char;
  res_file_type = array [ 0..3 ] of text;

{*************************************************************************

DIAGNOSTIC is a facility which when set true will output on the
terminal screen a list of all program procedures and functions called
by this program. This is used as a software development tool.

BASE is the pointer to the first entry in the program linked list.

 *************************************************************************}

VAR
  diagnostic : boolean;       { Diagnostic Facility           }
  base0 : link;               { base pointer for p0           }
  base1 : link;               { base pointer for p1           }
  base2 : link;               { base pointer for p2           }
  base3 : link;               { base pointer for p3           }
  p0 : link;                  { original linked list pointer  }
  p1 : link;                  { detection mechanism   "       }
  p2 : link;                  { boundary realocation   "      }
  p3 : link;                  { signature streaming   "       }
  user : text;
  trace_file : text;
  micro_file : text;
  code_file : text;
  result_file : res_file_type;
  analysis_file : text;
  analysis_file2 : text;
  area1 : integer;
  choice : boolean;
  response : restype;
  optimise_seeding : boolean;

{*************************************************************************

Function PLACEMENT_DEADLOCK determines whether an erroneous jump has its
generator and destination in the operands belonging to a single instruction.
The function returns TRUE if a placement deadlock is identified, otherwise
UNTRUE is returned.

 *************************************************************************}

FUNCTION placement_deadlock( p : link ): boolean;
VAR
  q : link;
  found : boolean;

BEGIN
  IF diagnostic THEN write( trace_file, '      placement_deadlock :');
  q := p;
  found := true;
  WHILE ((q^.address <> p^.jump_too^.address) AND (q <> nil) AND (found)) DO
    IF ( q^.address > p^.jump_too^.address ) THEN
      BEGIN {search backwards}
        IF ( q^.optype = OPCODE ) THEN
```

```
        found := false; {end of instruction search}
      q := q^.last_address;
    END { then }
    ELSE BEGIN {search forwards}
      IF ( q^.next_address^.optype = OPCODE ) THEN
        found := false; {end of instruction search}
      q := q^.next_address;
    END; { else }
  IF ( q = nil ) THEN found := false; {search extended outside used area}
  IF diagnostic THEN writeln( trace_file, found );
  placement_deadlock := found;
END;

{*****************************************************************************

Procedure DATA_ANALYSIS is used to analyse the data area code.

 *****************************************************************************}

PROCEDURE data_analysis( base : link );
VAR
  jump_out : integer;
  jump_in : integer;
  area : integer;
  seeds : integer;
  p : link;
  min, max : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'data_analysis' );
  area := 0;
  jump_out := 0;
  jump_in := 0;
  seeds := 0;
  p := base;
  min := base^.address;
  WHILE ( p <> nil ) DO
    BEGIN
      max := p^.address;
      p := p^.next_address;
    END; { while }
  p := base;
  WHILE ( p <> nil ) DO
    BEGIN
      area := area +2;
      IF ( p^.jump_address <> 0 ) THEN
        IF (( p^.jump_address + p^.address < min ) OR
          ( p^.jump_address + p^.address > max )) THEN
            jump_out := jump_out +1
          ELSE jump_in := jump_in +1;
      IF ( p^.op = SEED ) THEN seeds := seeds +1;
      p := p^.next_address;
    END; { while }
  write( analysis_file, 'INITIAL     ', area:4, '            ',
    jump_in:3, '              ', jump_out:3, '            ', seeds:3, '    ' );
  writeln( analysis_file, ' 0' );
END;

{*****************************************************************************
```



```
Procedure ANALYSIS is used to analyse the program area code.

 *****************************************************************************}

PROCEDURE analysis( title : titletype; var areal : integer ; base : link );
CONST
  OUTSIDE = 3;      {**********************}
  INSIDE = 4;       {*                    *}
  ODDADD = 5;       {*     JUMP STATES    *}
  INTERN = 6;       {*                    *}
  NOSEED = 7;       {*                    *}
  INFO = 8;         {**********************}
TYPE
  data_jump = array [ OUTSIDE..INFO ] of real;
  data_type = array [ OPERAND..OPCODE ] of data_jump;
  data_array = array [ 0..14 ] of data_type;
VAR
  p : link;
  data : data_array;
  number_seeds : real;
  unseeded_jumps : real;
  opcode_rests : real;
  oprand_rests : real;
  number_rests : real;
  opcode_jumps : real;
  oprand_jumps : real;
  number_jumps : real;
  opcode_retss : real;
  oprand_retss : real;
  number_retss : real;
  opcode_stops : real;
  oprand_stops : real;
  number_stops : real;
  recovery : real;
  area : integer;
  number_blocks : integer;
  area_size : real;
  i, j : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'analysis : ', title );
  number_seeds := 0;
  number_rests := 0;
  opcode_rests := 0;
  oprand_rests := 0;
  number_jumps := 0;
  opcode_jumps := 0;
  oprand_jumps := 0;
  number_retss := 0;
  opcode_retss := 0;
  oprand_retss := 0;
  number_stops := 0;
  opcode_stops := 0;
  oprand_stops := 0;
  recovery := 0;
  area := 0;
  number_blocks := 0;
  unseeded_jumps := 0;
  area_size := 0;
  p := base;
```

```
writeln( analysis_file );
FOR i := 0 TO 14 DO
  FOR j := OUTSIDE TO INFO DO
    BEGIN
      data[ i, OPCODE, j ] := 0;
      data[ i, OPERAND, j ] := 0;
    END;
WHILE ( p <> nil ) DO
  BEGIN
    data[ p^.jump_typo, p^.optypo, INFO ] := data[ p^.jump_typo, p^.optypo, INFO ]
    IF ( p^.jump_address mod 2 = 1 ) THEN
      data[ p^.jump_type, p^.optypo, ODDADD ] := data[ p^.jump_typo, p^.optypo, ODD
    IF (( p^.jump_address > 0 ) AND ( p^.jump_address mod 2 = 0 )) THEN
      BEGIN
        IF ( p^.jump_too <> nil ) THEN
          BEGIN
            IF placement_deadlock( p ) THEN
              data[ p^.jump_type, p^.optypo, INTERN ] := data[ p^.jump_typo, p^.opt
            data[ p^.jump_typo, p^.optypo, INSIDE ] := data[ p^.jump_typo, p^.optyp
            IF ( NOT p^.needed ) THEN
              data[ p^.jump_typo, p^.optypo, NOSEED ] := data[ p^.jump_typo, p^.opt
          END { if }
          ELSE data[ p^.jump_typo, p^.optypo, OUTSIDE ] := data[ p^.jump_typo, p^.o
      END { if }
      ELSE BEGIN
        { 50% prob odd address }
        data[ p^.jump_typo, p^.optypo, ODDADD ] := data[ p^.jump_typo, p^.optypo, O
        { Assume worst case    }
        data[ p^.jump_typo, p^.optypo, INSIDE ] := data[ p^.jump_typo, p^.optypo, I
      END; { else }
    IF ( p^.op = SEED ) THEN
      number_seeds := number_seeds +1;
    IF (( p^.op = BRA + 6 ) AND ( p^.next_address^.op = SEED )) THEN
      number_blocks := number_blocks +1;
    IF ( title = 'ORIGINAL CODE' ) THEN areal := areal +1;
    area := area +1;
    p := p^.next_address;
  END;
unseeded_jumps := data[ 1, OPCODE, NOSEED ] + data[ 1, OPERAND, NOSEED ] +
                  data[ 5, OPCODE, NOSEED ] + data[ 5, OPERAND, NOSEED ] +
                  data[ 6, OPCODE, NOSEED ] + data[ 6, OPERAND, NOSEED ];
opcode_rests := data[ 2, OPCODE, INFO ] + data[ 3, OPCODE, INFO ] + data[ 4, OPCODE
                data[ 7, OPCODE, INFO ] + data[ 12, OPCODE, INFO ] +
                data[ 13, OPCODE, INFO ] + data[ 14, OPCODE, INFO ];
oprand_rests := data[ 2, OPERAND, INFO ] + data[ 3, OPERAND, INFO  ] + data[ 4, OPE
                data[ 7, OPERAND, INFO ] + data[ 12, OPERAND, INFO ] +
                data[ 13, OPERAND, INFO ] + data[ 14, OPERAND, INFO ];
number_rests := opcode_rests + oprand_rests;
opcode_jumps := data[ 1, OPCODE, INFO ] + data[ 5, OPCODE, INFO ] + data[ 6, OPCODE
oprand_jumps := data[ 1, OPERAND, INFO ] + data[ 5, OPERAND, INFO ] + data[ 6, OPER
number_jumps := opcode_jumps + oprand_jumps;
opcode_retns := data[ 8, OPCODE, INFO ] + data[ 9, OPCODE, INFO ] + data[ 10, OPCOD
oprand_retns := data[ 8, OPERAND, INFO ] + data[ 9, OPERAND, INFO ] + data[ 10, OPE
number_retns := opcode_retns + oprand_retns;
opcode_stops := data[ 11, OPCODE, INFO ];
oprand_stops := data[ 11, OPERAND, INFO ];
number_stops := opcode_stops + oprand_stops;
IF ( area > 0 ) THEN
  recovery := ( number_seeds + area ) * 100 / ( area * 2 );
area_size := (( area / areal ) -1 ) *100;
```

```
write( analysis_file, title, '     ', ( area * 2 ):4, '             ',
  unseeded_jumps:3, '             ', number_blocks:3, '     ' );
IF ( area_size < 10 ) THEN write( analysis_file, '       ', area_size:2:3 )
  ELSE write( analysis_file, '      ', area_size:2:3 );
IF ( recovery < 10 ) THEN write( analysis_file, ' %          ', recovery:2:3 )
  ELSE write( analysis_file, ' %         ', recovery:2:3 );
writeln( analysis_file, ' %' );
writeln( analysis_file );
writeln( analysis_file );
writeln( analysis_file );
write( analysis_file2, title, '----------------------------OPCODE-----------------
writeln( analysis_file2, '----------------OPERAND-------------------------------
write( analysis_file2, '----------------OUT-----IN------ODD---WITHIN--NOSEED---TOT
writeln( analysis_file2, '-----|------OUT-----IN------ODD---WITHIN--NOSEED---TOTAL-
writeln( analysis_file2 );
write( analysis_file2, 'RESTART ----------------------------------------------');
write( analysis_file2, '------------------------------------------------');
writeln( analysis_file2, '------------------------------------------------');
writeln( analysis_file2 );
write( analysis_file2, 'CHK        ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 2, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 2, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'DIVS/DIVU  ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 3, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 3, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'ILLEGAL    ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 4, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 4, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'RESET      ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 7, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 7, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'TRAP       ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 12, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 12, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'TRAPV      ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 13, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 13, OPERAND, i ]:8:2 );
```

```
writeln( analysis_file2 );
write( analysis_file2, 'undefined  ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 14, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 14, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
writeln( analysis_file2 );
writeln( analysis_file2, 'TOTAL      ' );
writeln( analysis_file2 );
write( analysis_file2 , 'UNSPECIFIED JUMP ----------------------------------');
write( analysis_file2, '-----------------------------------------------');
writeln( analysis_file2, '-----------------------------------------------');
writeln( analysis_file2 );
write( analysis_file2, 'Bcc/BRA/BSR' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 1, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 1, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'JMP        ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 5, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 5, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'JSR        ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 6, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 6, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
writeln( analysis_file2 );
writeln( analysis_file2, 'TOTAL      ' );
writeln( analysis_file2 );
write( analysis_file2, 'RETURN ---------------------------------------------');
write( analysis_file2, '-----------------------------------------------');
writeln( analysis_file2, '-----------------------------------------------');
writeln( analysis_file2 );
write( analysis_file2, 'RTE        ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 8, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 8, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'RTR        ' );
FOR i := OUTSIDE TO INFO DO
write( analysis_file2, data[ 9, OPCODE, i ]:8:2 );
write( analysis_file2, '       |' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 9, OPERAND, i ]:8:2 );
writeln( analysis_file2 );
write( analysis_file2, 'RTS        ' );
FOR i := OUTSIDE TO INFO DO
  write( analysis_file2, data[ 10, OPCODE, i ]:8:2 );
```

```
  write( analysis_file2, '       |' );
  FOR i := OUTSIDE TO INFO DO
    write( analysis_file2, data[ 10, OPERAND, i ]:8:2 );
  writeln( analysis_file2 );
  writeln( analysis_file2 );
  writeln( analysis_file2, 'TOTAL      ' );
  writeln( analysis_file2 );
  write( analysis_file2, 'STOP/WAIT ------------------------------------------');
  write( analysis_file2, '-----------------------------------------------');
  writeln( analysis_file2, '-----------------------------------------------');
  writeln( analysis_file2 );
  write( analysis_file2, 'STOP       ' );
  FOR i := OUTSIDE TO INFO DO
    write( analysis_file2, data[ 11, OPCODE, i ]:8:2 );
  write( analysis_file2, '       |' );
  FOR i := OUTSIDE TO INFO DO
    write( analysis_file2, data[ 11, OPERAND, i ]:8:2 );
  writeln( analysis_file2 );
  writeln( analysis_file2 );
  writeln( analysis_file2, 'TOTAL      ' );
  writeln( analysis_file2 );
  writeln( analysis_file2 );
  writeln( analysis_file2 );
  writeln( analysis_file2 );
  writeln( analysis_file2 );
  writeln( analysis_file2 );
END;

{*****************************************************************************

Function JUMP_DIRECTION determines whether the erroneous jump results in
a positive ( FORWARD ) or negative ( BACKWARD ) offset to the program
counter. The START and FINISH positions through positive address increments
are determined for the exit and entry points in the program linked list
for the erroneous jump.

 *****************************************************************************}

FUNCTION jump_direction( var start, finish : link; prog_unit : link): boolean;
VAR
  i : integer;
  direction : boolean;

BEGIN
  IF diagnostic THEN write( trace_file, '    jump_direction :',
    prog_unit^.address:8, ' to ', prog_unit^.jump_too^.address:8 );
  start := prog_unit;
  finish := prog_unit;
  IF ( prog_unit^.address - START_ADDRESS
    < prog_unit^.jump_too^.address ) THEN
    BEGIN
      FOR i := 1 TO prog_unit^.jump_address DO
        IF ( i mod 2 = 0 ) THEN
          finish := finish^.next_address;
      direction := FORWARD;
    END
    ELSE BEGIN
      FOR i := prog_unit^.jump_address TO ( -1 ) DO
        IF ( i mod 2 = 0 ) THEN
          start := start^.last_address;
```

```
      direction := BACKWARD;
    END;
  IF diagnostic AND ( direction = FORWARD ) THEN writeln( trace_file, ' : forward ');
  IF diagnostic AND ( direction = BACKWARD ) THEN writeln( trace_file, ' : backward '
  jump_direction := direction;
END;

{*****************************************************************************

Procedure INTERVAL determines the number of erroneous jump generators and
destinations within an address range.

 *****************************************************************************}

PROCEDURE interval( var cross_over: integer; {No. Generators/Destinations}
                    start, finish : link );   {Address Range Boundaries   }
VAR
  p : link;          { Pointer to search interval  }

BEGIN
  IF diagnostic THEN write( trace_file, '    interval' );
  new( p );
  p := start;
  cross_over := 0;
  WHILE ( p <> finish ) DO
    BEGIN
      IF ( p <> start ) THEN
        IF ( p^.jump_from <> nil ) THEN
          IF (( NOT p^.jump_from^.seeded ) AND
             ( p^.jump_from^.optype = OPERAND )) THEN
            cross_over := cross_over +1;
      p := p^.next_address;
    END; { while }
  IF diagnostic AND ( cross_over > 0 ) THEN
    writeln( trace_file, ' : not clear (', cross_over:3, ' )');
  IF diagnostic AND ( cross_over = 0 ) THEN writeln( trace_file, ' : clear' );
END;

{*****************************************************************************

Function TEST_SEED determines whether the process of seeding is necessary
because there might already be a seed placed at the erroneous jump location.

 *****************************************************************************}

FUNCTION test_seed( unit : link; direction : boolean ): boolean;
VAR
  found : boolean;

BEGIN
  IF diagnostic THEN write( trace_file, '    test_seed' );
  found := false;
  CASE ( direction ) OF
      FORWARD :  found := unit^.seeded;
      BACKWARD : found := unit^.jump_from^.seeded;
    END; { case }
  IF diagnostic AND found THEN writeln( trace_file, ' : found');
  IF diagnostic AND NOT found THEN writeln( trace_file, ' : not found');
  test_seed := found;
END;
```

```
{*****************************************************************************

Procedure SEED_DETAILS determines the OFFSET required by the seed and
adjacent locations UNIT and FINISH inbetween which the seed is to be
placed. The appropriate SEEDED field for the jumping operand is set
true. Note that FINISH is modified due to determining the next or last
opcode in the program linked list.

 *****************************************************************************}

PROCEDURE seed_details( start : link;
                        direction : boolean;
                        var unit, finish : link;
                        var offset : integer );
BEGIN
  IF diagnostic THEN writeln( trace_file, '    seed_details' );
  offset := 1;
  CASE ( direction ) OF
      FORWARD :  BEGIN
                 start^.seeded := true;
                 unit := finish^.last_address;
                 WHILE ( unit^.optype = OPERAND ) DO
                   BEGIN
                     offset := offset +1;
                     unit := unit^.last_address;
                   END; { while }
                 END; { if }
      BACKWARD : BEGIN
                 finish^.seeded := true;
                 unit := start^.next_address;
                 WHILE ( unit^.optype = OPERAND ) DO
                   BEGIN
                     offset := offset +1;
                     unit := unit^.next_address;
                   END; { while }
                 END; { if }
    END;
  finish := unit;
  unit := unit^.last_address;
END;

{*****************************************************************************

Procedure PLACE_SEED inserts a seed of length OFFSET inbetween the
adjacent locations UNIT and FINISH in the program linked list.

 *****************************************************************************}

PROCEDURE place_seed( unit, finish : link; offset : integer);
VAR
  seed_block : link;
  q : link;
  i : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, '    place_seed' );
  new( q );
  q := unit;
  new( seed_block );
```

```
  seed_block^.optype := OPCODE;
  seed_block^.op := BRA + (offset *2);
  seed_block^.last_address := q;
  seed_block^.offset := 0;
  seed_block^.seeded := true;
  seed_block^.jump_address := (offset +1)*2; {accounting for opcode}
  seed_block^.jump_to := finish;
  finish^.jump_from := seed_block;
  seed_block^.jump_from := nil;
  seed_block^.jump_type := 1;    ( BRA )
  seed_block^.address := unit^.address +1;
  unit^.next_address := seed_block;
  q := seed_block;
  FOR i := 1 TO offset DO
    BEGIN
      new( seed_block );
      q^.next_address := seed_block;
      seed_block^.optype := OPERAND;
      seed_block^.op := SEED;
      seed_block^.last_address := q;
      seed_block^.offset := 0;
      seed_block^.seeded := true;
      seed_block^.jump_address := 1;
      seed_block^.jump_to := nil;
      seed_block^.jump_from := nil;
      seed_block^.jump_type := 1;   ( BRA : Exception Generating )
      seed_block^.address := unit^.address +i;
      q := seed_block;
    END; { for }
  q^.next_address := finish;
  finish^.last_address := q; { last seed_block }
  finish^.offset := offset;
END;

{*****************************************************************************

Function SEED_LOCATION checks whether there are any erroneous jump generat-
ors or destinations inbetween the present 'jump_to' location ( held by
PROG_UNIT ) and the associated 'jump_from' location for this particular
jump. The LEVEL of placement determines the size of group under attack.
If there are no erroneous jump generators or destinations in this interval
then a seed is placed accordingly. If an intersecting group of erroneous
jumps are identified then the first of that group is seeded and PROG_UNIT
is given the location terminting the group. This function returns TRUE for
a successful seed placement, otherwise a FALSE in returned.

If a group has any sub-groups of erroneous jump intersections then no
placement takes place.

*****************************************************************************}

FUNCTION seed_location( var prog_unit : link;
                        level : integer;
                        var marker : integer ): boolean;
VAR
  direction : boolean;      {Erroneous Jump Direction : Backwards/Forwards}
  start : link;             {Erroneous Jump Generator Location             }
  finish : link;            {Erroneous Jump Destination Location           }
  base : link;              {Low Address of Arroneous Jump Interval        }
  top : link;               {High Address of Erroneous Jump Interval       }
  unit : link;              {Code Unit For Seed Placement                  }
  offset : integer;         {No. Seeds in Detection Mechanism              }
  cross_over : integer;     {No. Generators/Destinations in Address Range  }
  placement : boolean;      {Seed Placement in code                        }

BEGIN
  IF diagnostic THEN writeln( trace_file, '  seed_location');
  new( start );
  new( finish );
  placement := FALSE;
  direction := jump_direction( start, finish, prog_unit );
  interval( cross_over, start, finish );
  CASE (direction) OF
      FORWARD :  BEGIN
                   base := start;
                   top  := finish;
                 END;
      BACKWARD : BEGIN
                   base := finish;
                   top  := start;
                 END;
    END; {case}
  IF ( diagnostic AND ( base^.address < marker )) THEN
    writeln( trace_file, '  => Group has sub-groups : No placement');
  IF (( cross_over = level ) AND NOT ( base^.address < marker )) THEN
    BEGIN
      marker := base^.address;
      IF ( NOT test_seed( prog_unit, direction )) THEN {require seeding}
        BEGIN
          prog_unit^.seeded := true;
          prog_unit := base;
          WHILE ((prog_unit <> nil) AND (prog_unit <> top)) DO
            BEGIN
              IF (prog_unit^.jump_to <> nil) THEN
                IF (prog_unit^.optype = OPERAND) THEN
                  IF (prog_unit^.jump_to^.address > top^.address) THEN
                    top := prog_unit^.jump_to;
              IF (prog_unit^.jump_from <> nil) THEN
                IF (prog_unit^.jump_from^.optype = OPERAND) THEN
                  IF (prog_unit^.jump_from^.address > top^.address) THEN
                    top := prog_unit^.jump_from;
              prog_unit := prog_unit^.next_address;
            END; {while}
          seed_details( start, direction, unit, finish, offset );
          IF ( NOT optimise_seeding ) THEN offset := 6;
          { offset := 6; Default for Koln Results }
          place_seed( unit, finish, offset );
          placement := true;
        END; {else}
    END; {if}
  seed_location := placement;
END;

{*****************************************************************************

Function DIGIT converts a character representing a hexidecimal digit into
an integer.

*****************************************************************************}
```

```
FUNCTION digit( ch : char ): integer;
VAR
  number : integer;

BEGIN
{ IF diagnostic THEN writeln( trace_file, '      digit' ); }
{ Diagnostic removed to reduce TRACE_FILE size             }

  number := 0;
  CASE ( ch ) OF
      '0' : number := 0;
      '1' : number := 1;
      '2' : number := 2;
      '3' : number := 3;
      '4' : number := 4;
      '5' : number := 5;
      '6' : number := 6;
      '7' : number := 7;
      '8' : number := 8;
      '9' : number := 9;
      'A' : number := 10;
      'B' : number := 11;
      'C' : number := 12;
      'D' : number := 13;
      'E' : number := 14;
      'F' : number := 15;
      'a' : number := 10;
      'b' : number := 11;
      'c' : number := 12;
      'd' : number := 13;
      'e' : number := 14;
      'f' : number := 15;
      ' ' : BEGIN { no word }
              number := -1;
            END;
      otherwise
            BEGIN { error }
              number := -2;
            END;
    END; { case }
  digit := number;
END;

{*****************************************************************************

Procedure BUILD_ADB constructs the linked list that represents the program.

 *****************************************************************************}

PROCEDURE build_adb;
CONST
  HEADER_LENGTH = 3;
VAR
  i : integer;
  q : link;                              { new item in linked list     }
  next_section : boolean;
  code : integer;
  old_code : integer;
  ch : char;
  number : integer;
  first : boolean;
  label1 : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'build_adb');
  p0 := base0;
  first := true;
  old_code := 0;
  i := 0;
  WHILE (( NOT eof( code_file )) AND ( i < HEADER_LENGTH )) DO
    BEGIN
      readln( code_file );
      i := i + 1;
    END; { while }
  next_section := false;
  label1 := 0;
  WHILE NOT next_section DO
    BEGIN { remove code }
      read( code_file, ch );
      WHILE NOT eoln( code_file ) DO
        BEGIN
          CASE ( ch ) OF
            ':'  : BEGIN { new label }
                     label1 := label1 +1;
                     read( code_file, ch );
                   END;
            ' '  : BEGIN { end of section }
                     next_section := true;
                     WHILE NOT eoln( code_file ) DO
                       read( code_file, ch );
                   END;
            '0'  : BEGIN { get data word }
                     read( code_file, ch ); { x }
                     IF ( ch = 'x' ) THEN
                       BEGIN
                         read( code_file, ch );
                         number := digit( ch );
                         code := 0;
                         WHILE ( number > -1 ) DO
                           BEGIN { build number }
                             code := ( code * 16 ) + number;
                             read( code_file, ch );
                             number := digit( ch );
                           END; { while }
                         new( q );
                         q^.jump_too := nil;   { DEFAULT }
                         q^.jump_from := nil;  { DEFAULT }
                         q^.op := code;
                         q^.optype := OPERAND;
                         q^.last_address := p0;
                         q^.needed := true;
                         q^.offset := label1;
                         IF first THEN
                           BEGIN { start of program }
                             q^.address := START_ADDRESS;
                             base0 := q;
                             first := false;
                           END { then }
                           ELSE BEGIN
                             q^.address := p0^.address +2;
```

```
                    p0^.next_address := q;
                  END; { else }
                q^.jump_address := 0;
                q^.jump_type := 0;   { NOT SET }
                q^.next_address := nil;
                p0 := q;
                read( code_file, ch );
              END; { if }
            END; { while }
          otherwise { due to unrecognisable code from adb }
            BEGIN { spaces }
              read( code_file, ch );
            END;
        END; { case }
      END; { while }
  END; { while first section }
next_section := false;
p0 := base0;
label1 := 0;
WHILE (( NOT eof( code_file )) AND ( NOT next_section )) DO
  BEGIN { second section : set opcode/operand }
    read( code_file, ch );
    WHILE (( NOT eoln( code_file )) AND ( NOT eof( code_file ))) DO
      BEGIN
        CASE ( ch ) OF
          '+' : BEGIN
                  read( code_file, ch ); { 0 }
                  read( code_file, ch ); { x }
                  code := 0;
                  read( code_file, ch );
                  number := digit( ch );
                  WHILE ( number > -1 ) DO
                    BEGIN
                      code := ( code * 16 ) + number;
                      read( code_file, ch );
                      number := digit( ch );
                    END; { while }
                  { character now ':' }
                  FOR i := 1 TO ( code - old_code ) DO
                    IF (( i mod 2 = 0 ) AND ( p0^.next_address <> nil )) THEN
                      p0 := p0^.next_address;
                  p0^.optype := OPCODE;
                  WHILE NOT eoln( code_file ) DO
                    read( code_file, ch );
                  old_code := code;
                  IF ( p0^.next_address = nil ) THEN
                    next_section := true;
                END;
          ':' : BEGIN { next label }
                  read( code_file, ch );
                  IF NOT eoln( code_file ) THEN
                    BEGIN
                      WHILE ( p0^.offset <> label1 +1 ) DO
                        p0 := p0^.next_address;
                      p0^.optype := OPCODE;
                      label1 := label1 +1;
                      old_code := 0;
                    END;
                  WHILE NOT eoln( code_file ) DO
                    read( code_file, ch );
                END;
          otherwise
                BEGIN
                  read( code_file, ch );
                END;
        END; { case }
      END; { while }
  END; { while : section section }
END;

{*****************************************************************************

Procedure BUILD_RNG constructs the linked list that represents a randomly
generated program.

 *****************************************************************************}

PROCEDURE build_rng( var q, base : link );
VAR
  p : link;
  i : integer;
  number : integer;
  length : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'build_rng');
  number := 16390; { random number to start process }
  new( q );
  q := base;
  writeln( 'Input decimal length of generated code : ');
  readln( user, length );
  FOR i := 1 TO length DO
    IF ( i mod 2 = 0 ) THEN
      BEGIN
        new( p );
        IF ( i = 2 ) THEN
          BEGIN
            p^.address := START_ADDRESS;
            base := p;
          END { then }
          ELSE BEGIN
            q^.next_address := p;
            p^.address := q^.address + 2;
          END; { else }
        p^.jump_too := nil;   { DEFAULT }
        p^.jump_from := nil;  { DEFAULT }
        p^.op := ( number * 13 ) mod ( 65536 + 1 );
        p^.optype := OPERAND;
        p^.needed := false;
        p^.offset := 0;
        p^.last_address := q;
        p^.next_address := nil;
        p^.jump_address := 0;
        p^.jump_type := 0;  { NOT SET }
        q := p;
        number := p^.op;
      END;
END;

{*****************************************************************************
```

```
Procedure BUILD_DIS constructs the linked list that represents the program.
BASE is a pointer to the first item in the list and is a global.

 ******************************************************************************}

PROCEDURE build_dis( var q, base : link );
VAR
  i : integer;
  count : integer;
  p : link;
  code : integer;
  ch : char;
  finish_line : boolean;
  first : boolean;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'build_dis' );
  i := 0;
  q := base;
  WHILE (( NOT eof( code_file ) ) AND ( i < HEADER_LENGTH )) DO
    BEGIN { remove header }
      readln( code_file );
      i := i +1;
    END; { while }
  first := true;
  WHILE ( NOT eof( code_file ) ) DO
    BEGIN { remove code }
      FOR i := 1 to INDENT DO
        read( code_file, ch );
      count := 0;
      finish_line := false;
      WHILE ( NOT finish_line ) DO
        BEGIN { get word }
          IF ( digit( ch ) > -1 ) THEN
            BEGIN
              new( p );
              code := 0;
              FOR i := 1 TO 4 DO
                BEGIN { get digits making word }
                  code := ( code * 16 ) + digit( ch );
                  read( code_file, ch );
                END;
              read( code_file, ch ); { perhaps next word }
              count := count +1;
              p^.op := code;
              p^.last_address := q;
              p^.offset := 0;
              IF ( count = 1 ) THEN p^.optype := OPCODE
                ELSE p^.optype := OPERAND;
              IF first THEN
                BEGIN
                  p^.address := START_ADDRESS;
                  base := p;
                  first := false;
                END { then }
                ELSE BEGIN
                  q^.next_address := p;
                  p^.address := q^.address +2;
                END; { else }
                p^.jump_too := nil;   { DEFAULT }
                p^.jump_from := nil;  { DEFAULT }
                p^.jump_type := 0;    { NOT SET }
                p^.next_address := nil;
                q := p;
              END { then }
              ELSE BEGIN { no more words }
                readln( code_file );
                finish_line := true;
              END; { else }
          END; { while }
      END; { while }
END;

{******************************************************************************

Procedure SET_JUMP constructs the jump data for a particular jump in the
linked list program.

 ******************************************************************************}

PROCEDURE set_jump( var p : link; lt, ll : integer );
VAR
  q : link;
  relative : integer;
  finish : boolean;

BEGIN
  IF diagnostic THEN write( trace_file, '   set_jump' );
  new( q );
  IF (( lt = 0 ) AND ( ll = 0 )) THEN
    BEGIN { displacement next word }
      IF diagnostic THEN writeln( trace_file, ' displacement next word' );
      IF ( p^.next_address^.op < MAX /2 ) THEN
        relative := p^.next_address^.op
        ELSE
        relative := p^.next_address^.op -MAX;
    END { then }
    ELSE BEGIN { displacement in lower byte }
      IF diagnostic THEN writeln( trace_file, ' displacement lower byte' );
      relative := p^.op mod 256;
      IF ( relative > 127 ) THEN
        relative := relative - 256;
    END; { else }
  relative := relative +2; { alter for PC pointing to first operand }
  q := p;
  finish := false;
  WHILE (( q^.address <> p^.address + relative ) AND ( NOT finish )) DO
    BEGIN
      IF ( relative < 0 ) THEN
        q := q^.last_address
        ELSE q := q^.next_address;
      IF ( q^.last_address = nil ) THEN finish := true;
      IF ( q^.next_address = nil ) THEN finish := true;
    END; { while }
  IF ( NOT finish ) THEN
    BEGIN
      p^.jump_too := q;
      p^.jump_too^.jump_from := p;
      {Default has set the SEEDED field to TRUE for all locations          }
```

```
      {Covers : out of area, opcode, placement deadlock, and already seeded}
      IF (( p^.optype = OPERAND ) AND ( NOT placement_deadlock( p )) AND
         ( q^.op <> SEED )) THEN {exception to default}
        p^.seeded := false;
    END; {if}
  p^.jump_address := relative;
END;

{*****************************************************************************

Procedure ALIGN_ADDRESS constructs jumps in the linked list program.

 *****************************************************************************}

PROCEDURE align_address( var p : link );
VAR
  q : link;
  i : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, '   align_address' );
  i := 0;
  q := p;
  WHILE (( i <> p^.jump_address ) AND ( q <> nil )) DO
    IF ( p^.jump_address > 0 ) THEN
      BEGIN
        q := q^.next_address;
        i := i +2;
      END { then }
      ELSE BEGIN
        q := q^.last_address;
        i := i -2;
      END; { else }
  p^.jump_too := q;
  p^.jump_too^.jump_from := p;
END;

{*****************************************************************************

Function INSTRU_SET tests an instruction to check whether it is a defined
instruction in  particular instruction set.

 *****************************************************************************}

FUNCTION instru_set( instru : integer ) : boolean;
TYPE
  digit_type = array [ 0..15 ] of char;

VAR
  length, match, num : integer;
  found : boolean;
  digits : digit_type;
  i, j : integer;

BEGIN
  IF diagnostic THEN write( trace_file, '   instru_set : ' );
  reset( micro_file, 'file=micro' );
  readln( micro_file, length );
  found := false;
  i := 1;
  WHILE (( i < length +1 ) AND ( NOT found )) DO
    BEGIN
      FOR j := 15 DOWNTO 0 DO
        read( micro_file, digits[ j ] );
      readln( micro_file );
      num := instru;
      match := 0;
      FOR j := 0 TO 15 DO
        BEGIN
          IF (( digit( digits[ j ] ) = num mod 2 ) OR ( digits[ j ] = 'X' )) THEN
            match := match +1;
          num := num div 2;
        END; { for }
      IF ( match = 16 ) THEN found := true;
      i := i +1;
    END; { while }
  IF diagnostic THEN writeln( trace_file, found );
  instru_set := found;
END;

{*****************************************************************************

Procedure BUILD_JUMPS constructs the legal and illegal jumps in the linked
list program.

 *****************************************************************************}

PROCEDURE build_jumps( var prog_unit : link ; base : link );
VAR
  tt, tl, lt, ll : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'build_jumps' );
  prog_unit := base;
  WHILE ( prog_unit <> nil ) DO
    BEGIN
      prog_unit^.jump_too := nil;
      prog_unit^.jump_from := nil;
      prog_unit^.jump_address := 0;
      prog_unit^.seeded := true;
      prog_unit := prog_unit^.next_address;
    END; {while}
  prog_unit := base;
  WHILE ( prog_unit <> nil ) DO
    BEGIN { set jumps }
      IF NOT instru_set( prog_unit^.op ) THEN
        prog_unit^.jump_type := 14;
      tt := prog_unit^.op div 4096;
      tl := prog_unit^.op mod 4096 div 256;
      lt := prog_unit^.op mod 256 div 16;
      ll := prog_unit^.op mod 16;
      CASE ( tt ) OF
          4 : BEGIN
                IF ( tl mod 2 = 1 ) THEN
                  BEGIN
                    CASE ( lt ) OF
                        8 : BEGIN
                              IF ( ll < 8 ) THEN
                                BEGIN { JMP : Dn }
                                  prog_unit^.jump_type := 2;
```

```
                END;
            END;
        9 : BEGIN { JMP : (An)/(An)+ }
              prog_unit^.jump_typo := 2;
            END;
       10 : BEGIN { CHK : d(An)/-(An) }
              prog_unit^.jump_type := 2;
            END;
       11 : BEGIN
              IF ( ll < 13 ) THEN
                BEGIN { CHK }
                  prog_unit^.jump_type := 2;
                END;
            END;
      otherwise
            BEGIN
              { no action }
            END;
     END; { case }
  END; { if }
CASE ( tl ) OF
   10 : BEGIN
          IF (( lt = 15 ) AND ( ll = 12 )) THEN
            prog_unit^.jump_type := 4; { ILLEGAL }
        END;
   14 : BEGIN
          CASE ( lt ) OF
              4 : BEGIN
                    prog_unit^.jump_typo := 12; { TRAP }
                  END;
              7 : BEGIN
                    CASE ( ll ) OF
                        0 : BEGIN { RESET }
                              prog_unit^.jump_typo := 7;
                            END;
                        2 : BEGIN { STOP }'
                              prog_unit^.jump_typo := 11;
                            END;
                        3 : BEGIN { RTE }
                              prog_unit^.jump_typo := 8;
                            END;
                        5 : BEGIN { RTS }
                              prog_unit^.jump_typo := 10;
                            END;
                        6 : BEGIN { TRAPV }
                              prog_unit^.jump_type := 13;
                            END;
                        7 : BEGIN { RTR }
                              prog_unit^.jump_typo := 9;
                            END;
                       otherwise
                            BEGIN
                              { no action }
                            END;
                    END; { case }
                  END;
              9 : BEGIN
                    IF ( ll < 8 ) THEN
                      BEGIN { JSR : (An) }
                        prog_unit^.jump_typo := 6;
```



```
                      END;
                  END;
             10 : BEGIN
                    IF ( ll > 7 ) THEN
                      BEGIN { JSR : d(An) }
                        prog_unit^.jump_typo := 6;
                      END;
                  END;
             11 : BEGIN
                    CASE ( ll ) OF
                        0 : BEGIN { JSR : d(A0,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                        1 : BEGIN { JSR : d(A1,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                        2 : BEGIN { JSR : d(A2,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                        3 : BEGIN { JSR : d(A3,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                        4 : BEGIN { JSR : d(A4,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                        5 : BEGIN { JSR : d(A5,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                        6 : BEGIN { JSR : d(A6,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                        7 : BEGIN { JSR : d(A7,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                        8 : BEGIN { JSR : Abs.W }
                              prog_unit^.jump_type := 6;
                              prog_unit^.jump_address :=
                                prog_unit^.next_address^.op -
                                prog_unit^.address +
                                START_ADDRESS;
                              align_address( prog_unit );
                            END;
                        9 : BEGIN { JSR : Abs.L }
                              prog_unit^.jump_type := 6;
                              prog_unit^.jump_address :=
                                prog_unit^.next_address^.op * 66536 +
                                prog_unit^.next_address^.next_address
                                prog_unit^.address +
                                START_ADDRESS;
                              align_address( prog_unit );
                            END;
                       10 : BEGIN { JSR : d(PC) }
                              prog_unit^.jump_type := 6;
                              set_jump( prog_unit, 0, 0 );
                            END;
                       11 : BEGIN { JSR : d(PC,Xi) }
                              prog_unit^.jump_type := 6;
                            END;
                      otherwise
                            BEGIN
```

```
                { no action }
              END;
        END; { case }
    END;
13 : BEGIN
      IF ( ll < 8 ) THEN
        BEGIN { JMP : (An) }
          prog_unit^.jump_type := 5;
        END;
    END;
14 : BEGIN
      IF ( ll > 7 ) THEN
        BEGIN { JMP : d(An) }
          prog_unit^.jump_type := 5;
        END;
    END;
15 : BEGIN
      CASE ( ll ) OF
          0 : BEGIN { JMP : d(A0,Xi) }
                prog_unit^.jump_type := 5;
              END;
          1 : BEGIN { JMP : d(A1,Xi) }
                prog_unit^.jump_type := 5;
              END;
          2 : BEGIN { JMP : d(A2,Xi) }
                prog_unit^.jump_type := 5;
              END;
          3 : BEGIN { JMP : d(A3,Xi) }
                prog_unit^.jump_type := 5;
              END;
          4 : BEGIN { JMP : d(A4,Xi) }
                prog_unit^.jump_type := 5;
              END;
          5 : BEGIN { JMP : d(A5,Xi) }
                prog_unit^.jump_type := 5;
              END;
          6 : BEGIN { JMP : d(A6,Xi) }
                prog_unit^.jump_type := 5;
              END;
          7 : BEGIN { JMP : d(A7,Xi) }
                prog_unit^.jump_type := 5;
              END;
          8 : BEGIN { JMP : Abs.W }
                prog_unit^.jump_type := 5;
                prog_unit^.jump_address :=
                  prog_unit^.next_address^.op -
                  prog_unit^.address +
                  START_ADDRESS;
                align_address( prog_unit );
              END;
          9 : BEGIN { JMP : Abs.L }
                prog_unit^.jump_type := 5;
                prog_unit^.jump_address :=
                  prog_unit^.next_address^.op * 65536 +
                  prog_unit^.next_address^.next_address
                  prog_unit^.address +
                  START_ADDRESS;
                align_address( prog_unit );
              END;
         10 : BEGIN { JMP : d(PC) }
                                    prog_unit^.jump_type := 5;
                                    set_jump( prog_unit, 0, 0 );
                                  END;
                             11 : BEGIN { JMP : d(PC,Xi) }
                                    prog_unit^.jump_type := 5;
                                  END;
                           otherwise
                                  BEGIN
                                    { no action }
                                  END;
                         END; { case }
                       END;
                  otherwise
                       BEGIN
                         { no action }
                       END;
                  END; { case }
             END;
        otherwise
             BEGIN
               { no action }
             END;
        END; { case }
    END;
6 : BEGIN { Bcc/BRA/BSR }
      prog_unit^.jump_type := 1;
      CASE ( tl ) OF
          0 : BEGIN { BRA }
                set_jump( prog_unit, lt, ll );
              END;
          1 : BEGIN { not a jump }
                { no action }
              END;
        otherwise { BCC }
              BEGIN
                set_jump( prog_unit, lt, ll );
              END;
        END; { case }
    END;
8 : BEGIN
      CASE ( lt ) OF
         12 : BEGIN
                IF ( ll < 8 ) THEN { DIVS/DIVU }
                  prog_unit^.jump_type := 3;
              END;
         13 : BEGIN { DIVS/DIVU }
                prog_unit^.jump_type := 3;
              END;
         14 : BEGIN { DIVS/DIVU }
                prog_unit^.jump_type := 3;
              END;
         15 : BEGIN
                IF ( ll < 16 ) THEN { DIVS/DIVU }
                  prog_unit^.jump_type := 3;
              END;
       otherwise
              BEGIN
                { no action }
              END;
      END; { case }
```

```
            END;
          otherwise
            BEGIN
              { no action }
            END;
        END; { case }
      prog_unit := prog_unit^.next_address;
    END; { while }
END;

(*****************************************************************************

Procedure COPY_LIST takes a copy of the a program linked list.

 *****************************************************************************)

PROCEDURE copy_list( var prog_unit, base : link ; list_base : link );
VAR
  p, q : link;
  first : boolean;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'copy_list');
  prog_unit := base;
  p := list_base;
  q := nil;
  first := true;
  WHILE ( p <> nil ) DO
    BEGIN
      new( prog_unit );
      prog_unit^ := p^;
      prog_unit^.last_address := q;
      prog_unit^.jump_too := nil;
      prog_unit^.jump_from := nil;
      IF first THEN
        BEGIN
          base := prog_unit;
          first := false;
        END { if }
        ELSE BEGIN { tie up pointers }
          prog_unit^.last_address^.next_address := prog_unit;
        END; { else }
      q := prog_unit;
      p := p^.next_address;
    END; { while }
END;


(*****************************************************************************

Procedure ADDRESS_LIST walks through the program linked list starting at
BASE and assigns in a consecutive fashion addresses, commencing at the
passed parameter START_ADDRESS, to the new seeded linked list.

 *****************************************************************************)

PROCEDURE address_list( var prog_unit : link;
                        base : link ;
                        start_address : integer );
VAR
  i : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'address_list');
  prog_unit := base;
  i := start_address;
  WHILE ( prog_unit <> nil ) DO
    BEGIN
      prog_unit^.address := i;
      i := i + 2;
      prog_unit := prog_unit^.next_address;
    END;
END;

(*****************************************************************************

Procedure SEED_PLACEMENT injects seeds through a number of passes through
code) to resolve the placement of optimum size detection mechanisms to
cover all erroneous jumps. The parameter PLACEMENT keeps track of
successful seed insertions for the recursive LEVEL of code interrigation.

 *****************************************************************************)

PROCEDURE seed_placement( base1: link;
                          level: integer;
                          placement: boolean );

VAR
  scan : integer;          { Recursive parameter representing level }
  unit : link;             { Code unit under investigation          }
  altered : boolean;       { Seeds insertion                        }
  incomplete : boolean;    { Seeds inserted at this level           }
  marker : integer;        { Low address of last seeded interval    }

BEGIN
  IF diagnostic THEN writeln( trace_file, '  seed_placement (LEVEL',level:2,')');
  incomplete := true;
  WHILE incomplete DO
    BEGIN
      incomplete := false;
      marker := -1;       { Set Up Default }
      unit := base1;
      WHILE ( unit <> nil ) DO
        IF ( NOT unit^.seeded  ) THEN
          BEGIN
            IF ( marker = -1 ) THEN marker := unit^.address;
            altered := seed_location( unit, level, marker );
            incomplete := incomplete OR altered;
            placement := placement OR altered;
            IF ( NOT altered ) THEN unit := unit^.next_address;
          END {if}
          ELSE unit := unit^.next_address;
      IF diagnostic THEN writeln( trace_file, '  INFO : Pass Complete');
      unit := base1;
      WHILE (( unit <> nil ) AND ( unit^.seeded )) DO
        unit := unit^.next_address;
      IF ( unit <> nil ) THEN
        BEGIN
          address_list( unit, base1, base1^.address );
          build_jumps( unit, base1 );
```

```
      END {if}
      ELSE incomplete := false;
    END; { while }
  IF ( level > 0 ) THEN
    FOR scan := 0 TO (level -1) DO
      seed_placement( base1, scan, placement );
END;

{*****************************************************************************

Procedure SEED_LIST repeatedly goes through the linked program list until
all erroneous jumps are resolved by seeding. The number of passes required
is dependent on the complexity of invalid branch groups.

 *****************************************************************************}

PROCEDURE seed_list( var p1, base1 : link );
VAR
  complete : boolean;     { Seed placements complete in code pass          }
  placement : boolean;    { Seed placements made in code pass              }
  level : integer;        { No. intersecting erroneous jumps in group      }

BEGIN
  IF diagnostic THEN writeln( trace_file, 'seed_list' );
  complete := false;
  level :=0;
  WHILE NOT complete DO
    BEGIN
      complete := true;
      placement := false;
      seed_placement( base1, level, placement );
      p1 := base1;
      WHILE ( p1 <> nil ) DO
        BEGIN
          IF ( NOT p1^.seeded  ) THEN complete := false;
          p1 := p1^.next_address;
        END; {while}
      IF (NOT complete AND NOT placement ) THEN level := level +1;
    END; { while }
END; { begin }

{*****************************************************************************

Procedure RELOCATE_LIST walks through the linked program list relocating
each instruction at a 8-word address boundary. Specifically 8-word because
the 68000 max instruction is 5-words.

 *****************************************************************************}

PROCEDURE relocate_list;
CONST
  BOUNDARY = 16;
VAR
  i : integer;          { instruction byte/word index  }
  p, q, r : link;       { program linked list pointer  }
  first : boolean;
  finish : boolean;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'relocate_list' );
  q := nil;
  new( r ); { NOP }
    r^.jump_too := nil;       { DEFAULT }
    r^.jump_from := nil;      { DEFAULT }
    r^.address := 0;          { NOT SET }
    r^.optype := OPCODE;      { SET     }
    r^.op := 20081;           { SET     }
    r^.needed := false;       { DEFAULT }
    r^.offset := 0;           { DEFAULT }
    r^.jump_address := 0;     { DEFAULT }
    r^.jump_type := 0;        { NOT SET }
    r^.last_address := nil;   { NOT SET }
    r^.next_address := nil;   { NOT SET }
  p2 := base2;
  first := true;
  q := nil;
  WHILE ( p2 <> nil ) DO
    BEGIN
      i := 2; { opcode }
      p2^.last_address := q;
      p2 := p2^.next_address;
      finish := false;
      WHILE (( p2 <> nil ) AND NOT finish ) DO
        IF ( p2^.optype <> OPERAND ) THEN
          finish := true
          ELSE BEGIN
            i := i +2;
            p2 := p2^.next_address;
          END; { else }
      IF ( p2 <> nil ) THEN
        BEGIN
          p := p2;
          q := p2^.last_address;
          WHILE ( i < BOUNDARY ) DO
            BEGIN
              i := i +2; { NOP }
              new( p2 );
              p2^ := r^;
              p2^.last_address := q;
              p2^.last_address^.next_address := p2;
              q := p2;
            END; { while }
          p2 := p;
          q^.next_address := p2;
        END; { if }
    END; { while }
END;

{*****************************************************************************

Procedure STREAM_LIST walks through the linked program list until hashing
blocks are inserted.

 *****************************************************************************}

PROCEDURE stream_list;
CONST
  HASH_NUMBER = 1;  { Number of units required for hashing }
  RANDOM_GENERATOR = 16390;
VAR
```

```
  p, q, r : link;
  i : integer;
  finish : boolean;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'stream_list' );
  q := nil;
  new( r ); { NOP }
    r^.jump_too := nil;          { DEFAULT }
    r^.jump_from := nil;         { DEFAULT }
    r^.address := 0;             { NOT SET }
    r^.op := RANDOM_GENERATOR;   { NOT SET }
    r^.optype := OPERAND;        { SET     }
    r^.needed := true;           { DEFAULT }
    r^.offset := 0;              { DEFAULT }
    r^.jump_address := 0;        { DEFAULT }
    r^.jump_type := 0;           { NOT SET }
    r^.last_address := nil;      { NOT SET }
    r^.next_address := nil;      { NOT SET }
  p3 := base3;
  WHILE ( p3 <> nil ) DO
    BEGIN
      p3^.last_address := q;
      IF (( p3^.jump_type <> 0 ) AND ( p3^.optype = OPCODE )) THEN
        BEGIN
          p3 := p3^.next_address;
          finish := false;
          WHILE (( p3 <> nil ) AND NOT finish ) DO
            IF ( p3^.optype <> OPERAND ) THEN
              finish := true
              ELSE p3 := p3^.next_address;
          IF ( p3 <> nil ) THEN
            BEGIN
              p := p3;
              q := p3^.last_address;
              FOR i := 1 TO HASH_NUMBER DO
                BEGIN
                  r^.op := ( r^.op * 13 ) mod ( 65536 +1 );
                  new( p3 );
                  p3^ := r^;
                  p3^.last_address := q;
                  p3^.last_address^.next_address := p3;
                  q := p3;
                END; { for }
              p3 := p;
              q^.next_address := p3;
            END; { if }
        END { if }
        ELSE p3 := p3^.next_address;
    END; { while }
END; { begin }

{*****************************************************************************

Procedure STANDARD_OFFSET alters an instructions jump too address data
due to the offset introduced by the placement of the needs.

 *****************************************************************************}

PROCEDURE standard_offset( lt, ll : integer; var p : link );

VAR
  address_offset : integer;

BEGIN
  IF diagnostic THEN write( trace_file, '     standard_offset' );
  address_offset := p^.jump_too^.address - p^.address;
  p^.jump_address := address_offset;
  address_offset := address_offset -2; { allow for PC offset }
  IF ( address_offset < 0 ) THEN
    address_offset := MAX + address_offset;
  IF (( lt = 0 ) AND ( ll = 0 )) THEN
    BEGIN { alter next word }
      p^.next_address^.op := address_offset;
    END { then }
    ELSE BEGIN { alter lower byte }
      p^.op := ( p^.op div 256 ) + address_offset;
    END; { else }
  IF diagnostic THEN
    IF ( address_offset > MAX ) THEN
      writeln( trace_file, '   ERROR : offset overflow at ', p^.address :6 )
      ELSE IF ( address_offset > -1 ) THEN
        writeln( trace_file )
        ELSE { address_offset < 0 }
          writeln( trace_file, '   ERROR : offset underflow at ', p^.address :6 );
END;

{*****************************************************************************

Procedure ALIGN_LIST re-aligns all opcode jumps. This is necessary due
to the placement of needs within the code. No allowance has been made
for branch offset overflow by this procedure.

 *****************************************************************************}

PROCEDURE align_list( var prog_unit : link ; base : link );
VAR
  tt, tl, lt, ll : integer;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'align_list');
  prog_unit := base;
  WHILE ( prog_unit <> nil ) DO
    BEGIN { not end of linked list }
      IF (( prog_unit^.optype = OPCODE ) AND
         ( prog_unit^.jump_too <> nil )) THEN
       IF ((( prog_unit^.next_address <> nil ) AND
          ( prog_unit^.next_address^.op <> SEED )) OR
          ( prog_unit^.next_address = nil )) THEN
         BEGIN { jump needs to be re-aligned }
          tt := prog_unit^.op div 4096;
          tl := prog_unit^.op mod 4096 div 256;
          lt := prog_unit^.op mod 256 div 16;
          ll := prog_unit^.op mod 16;
          CASE ( tt ) OF
              4 : BEGIN
                    IF ( tl = 14 ) THEN
                      CASE ( lt ) OF
                          11 : BEGIN
                                 IF ( ll = 10 ) THEN { JSR : d(PC) }
                                   standard_offset( 0, 0, prog_unit );
```

```
                END;
            15 : BEGIN
                   IF ( l1 = 10 ) THEN { JMP : d(PC) }
                     standard_offset( 0, 0, prog_unit );
                END;
           otherwise
                BEGIN
                  { no action }
                END;
          END; { case }
      END;
  6 : BEGIN
        CASE ( tl ) OF
          0 : BEGIN { BRA }
                IF ( NOT (( lt = 0 ) AND ( l1 = 6 )) AND
                  ( prog_unit^.next_address^.op = SEED )) THEN { not so
                  standard_offset( lt, l1, prog_unit );
                END;
          1 : BEGIN { not a jump }
                { no action }
                END;
          otherwise { BCC }
                BEGIN { Bcc : tl = 2 to 15 }
                  standard_offset( lt, l1, prog_unit );
                END;
        END; { case }
    END;
  otherwise
      BEGIN
        { no action }
      END;
END; { case }
CASE ( prog_unit^.op ) OF { check absolute jumps }
20152 : BEGIN { JSR : Abs.W }
          writeln( trace_file, '     abs address');
          prog_unit^.jump_address :=
            prog_unit^.jump_too^.address -
            prog_unit^.address;
          prog_unit^.next_address^.op :=
            prog_unit^.jump_too^.address;
        END;
20153 : BEGIN { JSR : Abs.L }
          writeln( trace_file, '     abs address');
          prog_unit^.jump_address :=
            prog_unit^.jump_too^.address -
            prog_unit^.address;
          prog_unit^.next_address^.op :=
            prog_unit^.jump_too^.address div 66536;
          prog_unit^.next_address^.next_address^.op :=
            prog_unit^.jump_too^.address mod 66536;
        END;
20216 : BEGIN { JMP : Abs.W }
          writeln( trace_file, '     abs address');
          prog_unit^.jump_address :=
            prog_unit^.jump_too^.address -
            prog_unit^.address;
          prog_unit^.next_address^.op :=
            prog_unit^.jump_too^.address;
        END;
20217 : BEGIN { JMP : Abs.L }
          writeln( trace_file, '     abs address');
          prog_unit^.jump_address :=
            prog_unit^.jump_too^.address -
            prog_unit^.address;
          prog_unit^.next_address^.op :=
            prog_unit^.jump_too^.address div 66536;
          prog_unit^.next_address^.next_address^.op :=
            prog_unit^.jump_too^.address mod 66536;
        END;
      otherwise
          BEGIN
            { no action }
          END;
    END; { case }
  END; { if }
  prog_unit := prog_unit^.next_address;
 END; { while }
END;

{*****************************************************************************

Procedure TIDY_LIST walks through the enhanced coding (presently detection
mechanism placement) and combines any detectin mechanisms placed as direct
neighbours. The father of this procedure will call ALIGN_LIST to re-align
the combined detection mechanism 'hop_over' jump.

 *****************************************************************************}

PROCEDURE tidy_list( var p1, base1 : link );
VAR
   q1, q2 : link;
   mechanism : boolean;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'tidy_list' );
  p1 := base1;
  WHILE ( p1 <> nil ) DO
    BEGIN
      IF (( p1^.optype = OPCODE ) AND ( p1^.op > 24576 ) AND
      ( p1^.op < 24588 )) THEN {BRA}
        BEGIN
          q1 := p1; { starting point }
          p1 := p1^.next_address;
          mechanism := true;
          WHILE (( p1^.optype = OPERAND ) AND ( p1 <> nil )) DO
            BEGIN
              IF ( p1^.op <> SEED ) THEN mechanism := false;
              p1 := p1^.next_address;
            END; {while}
          q2 := p1; { next starting point for next search }
          IF (( p1^.op > 24576 ) AND ( p1^.op < 24588 )) THEN {BRA}
          IF ( mechanism ) THEN { check for adjacent mechanism }
            BEGIN
              p1 := p1^.next_address; { operand }
              mechanism := true;
              WHILE (( p1^.optype = OPERAND ) AND ( p1 <> nil )) DO
                BEGIN
                  IF ( p1^.op <> SEED ) THEN mechanism := false;
                  p1 := p1^.next_address;
                END; {while}
```

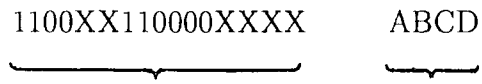
```
                IF ( mechanism ) THEN ( found adjacent mechanisms )
                  BEGIN
                    q1^.jump_address := q1^.jump_address + q2^.jump_address;
                    q1^.op := q1^.op + q2^.jump_address;
                    q1^.jump_too := p1;
                    q2^.offset := 0;
                    q2^.jump_too^.jump_from := q1;
                    q2^.jump_too := nil;
                    q2^.op := SEED;
                    q2^.optypo := OPERAND; {SEED}
                    q2^.jump_typo := 12;   {TRAP}
                    q2^.jump_address := 1; {SEED}
                    p1^.jump_from := q1;
                    p1 := p1^.last_address;
                  END; {if}
              END; {if}
          END; {if}
        p1 := p1^.next_address;
      END; {while}
END;

{*****************************************************************************

 Procedure PRINT_LIST outputs the linked list fields of address, opcode or
 operand, contents of location, and where appropriate the location pointed
 to if the present instruction is a jump.

 *****************************************************************************}

PROCEDURE print_list( num : integer ; title : titletypo );
VAR
  prog_unit : link;

BEGIN
  IF diagnostic THEN writeln( trace_file, 'print_list' );
  CASE ( num ) OF
      0 : BEGIN ( Original Code )
            prog_unit := base0;
          END;
      1 : BEGIN ( Detection Mechanism Placement )
            prog_unit := base1;
          END;
      2 : BEGIN ( Boundary Realocation )
            prog_unit := base2;
          END;
      3 : BEGIN ( Instruction Stream Signatures )
            prog_unit := base3;
          END;
      otherwise
          BEGIN
            { no action }
          END;
    END; { case }
  writeln( result_file[ num ] );
  writeln( result_file[ num ], '-------------------------------------------------------
  writeln( result_file[ num ] );
  writeln( result_file[ num ], title, ' CODE LISTING');
  writeln( result_file[ num ] );
  WHILE ( prog_unit <> nil ) DO
    BEGIN
      write( result_file[ num ], prog_unit^.address, prog_unit^.optypo,
        prog_unit^.op, prog_unit^.seed2d);
      IF ( prog_unit^.jump_address <> 0 ) THEN
        IF ( prog_unit^.jump_address mod 2 = 0 ) THEN
          writeln( result_file[ num ], prog_unit^.jump_address)
          ELSE writeln( result_file[ num ], '   Exception' )
        ELSE writeln( result_file[ num ] );
      prog_unit := prog_unit^.next_address;
    END; { while }
END;

{*****************************************************************************

 The MAIN program starts here, and initially constructs a linked list from
 the disassembly output from 'dis' ( Vitesse 5 System, Engineering Dept,
 University of Durham ), then places seeds throughout where required, then
 re-aligns any called addresses within the program list, and then finally
 tidys the program list. The completed program list is output into a file
 'results' in a coding type format.

 *****************************************************************************}

 BEGIN
  diagnostic := true;                                  { Diagnostic Facility ON }
  rewrite( trace_file, 'file=trace' );                 { Special mts command    }
  base0 := nil;                                        { Set default            }
  base1 := nil;                                        { Set default            }
  base2 := nil;                                        { Set default            }
  base3 := nil;                                        { Set default            }
  p0 := base0;                                         { SET default            }
  p1 := base1;                                         { SET default            }
  p2 := base2;                                         { SET default            }
  p3 := base3;                                         { SET default            }
  writeln( trace_file, 'TRACE FOR PARUT ');
  writeln( trace_file, '================');
  writeln( trace_file );
  IF diagnostic THEN writeln( trace_file, 'program' );
  rewrite( result_file[0], 'file=PARUTres0' );         { Special mts command }
  rewrite( result_file[1], 'file=PARUTres1' );         { Special mts command }
  rewrite( result_file[2], 'file=PARUTres2' );         { Special mts command }
  rewrite( result_file[3], 'file=PARUTres3' );         { Special mts command }
  rewrite( analysis_file, 'file=analysis' );           { Special mts command }
  rewrite( analysis_file2, 'file=analysis2' );         { Special mts command }
  reset( code_file, 'file=code' );                     { Special mts command }
  reset( user, 'file=*MSOURCE*,INTERACTIVE' );
  areal := 0;
  choice := false;
  writeln( 'PARUT : TRANSIENT FAULT RECOVERY TOOL');
  writeln( '=====================================');
  writeln; writeln;
  writeln('INFORMATION INPUT :- Please type appropriate response');
  writeln; writeln;
  WHILE NOT choice DO
    BEGIN
      writeln('Data or Program Area (DATA/PROGRAM) ?');
      readln( user, response );
      IF ( response = 'DATA' ) THEN
        BEGIN
          WHILE NOT choice DO
            BEGIN
```

```
        writeln('Randomly Generated or Code Input ?');
        readln( user, response );
        IF ( response = 'RANDOM' ) THEN
          BEGIN
            choice := true;
            build_rng( p0, base0 );
            write( analysis_file, ' RANDOMLY GENERATED ' );
          END; { if }
        IF ( response = 'CODE' ) THEN
          BEGIN
            choice := true;
            build_adb;
            write( analysis_file, ' CODED ' );
          END; { if }
      END; { while }
    writeln( analysis_file, 'DATA AREA : ANALYSIS' );
    writeln( analysis_file, '=============================================',
      '==========');
    writeln( analysis_file );
    writeln( analysis_file, ' TYPE    PROGRAM  AREA   ',
      'JUMPS INSIDE   JUMPS OUTSIDE    NO. SEEDS    RECOVERY FACTOR' );
    writeln( analysis_file, '          ( BYTES )     ',
      ' DATA AREA        DATA AREA       ');
    writeln( analysis_file );
    build_jumps( p0, base0 );
    print_list( 0, 'ORIGINAL' );
    data_analysis( base0 );
  END; { if }
IF ( response = 'PROGRA' ) THEN
  BEGIN
    writeln( analysis_file, ' PROGRAM AREA : ANALYSIS' );
    writeln( analysis_file, '=============================================',
      '==========');
    writeln( analysis_file );
    writeln( analysis_file, ' TYPE    PROGRAM  AREA   ',
      'NO.  UNSEEDED   NO. SEED BLOCKS  ',
      ' AREA INCREASE');
    writeln( analysis_file, '          ( BYTES )     ',
      'OPERAND JUMPS ');
    writeln( analysis_file );
    build_adb;
    response := 'NO';
    writeln('Detection Mechanism Placement (YES/NO)?');
    readln( user, response );
    IF ( response = 'YES' ) THEN
       BEGIN
          copy_list( p1, base1, base0 );
          build_jumps( p1, base1 );
          response := 'NO';
          writeln('=> Optimise Placement (YES/NO)?');
          readln( user, response );
          IF ( response = 'YES' ) THEN optimise_seeding := true
            ELSE optimise_seeding := false;
          seed_list( p1, base1 );
          tidy_list( p1, base1 );
          address_list( p1, base1, START_ADDRESS );
          align_list( p1, base1 );
          print_list( 1, 'DETECTION MECHANISM PLACEMENT' );
       END; {if}
    response := 'NO';
    writeln('Boundary Relocation (YES/NO)?');
    readln( user, response );
    IF ( response = 'YES' ) THEN
       BEGIN
          copy_list( p2, base2, base0 );
          build_jumps( p2, base2 );
          relocate_list;
          address_list( p2, base2, START_ADDRESS );
          align_list( p2, base2 );
          print_list( 2, 'BOUNDARY RELOCATION' );
       END; {if}
    response := 'NO';
    writeln('Signature Placement (YES/NO)?');
    readln( user, response );
    IF ( response = 'YES' ) THEN
       BEGIN
          copy_list( p3, base3, base0 );
          build_jumps( p3, base3 );
          stream_list;
          address_list( p3, base3, START_ADDRESS );
          align_list( p3, base3 );
          print_list( 3, 'INSTRUCTION STREAM SIGNATURES' );
       END; {if}
    writeln;
    writeln('<<< Original Code (for comparison) being prepared >>>');
    writeln;
    build_jumps( p0, base0 );
    address_list( p0, base0, START_ADDRESS );
    print_list( 0, 'ORIGINAL' );
    response := 'NO';
    writeln('Analysis Required (YES/NO)?');
    readln( user, response );
    IF ( response = 'YES' ) THEN
       BEGIN
          analysis( 'ORIGINAL CODE', area1, base0 );
          analysis( 'DETECTION MECHANISM PLACEMENT', area1, base1 );
       END; {if}
    choice := true;
    writeln( analysis_file );
    writeln( analysis_file );
    writeln( analysis_file, '() = The subset number of jumps that are');
    writeln( analysis_file, '     outside the software block : Opcode jumps');
    writeln( analysis_file, '     may be assumed to be correctly aligned ;');
    writeln( analysis_file, '     Operand jumps may be odd address exception');
    writeln( analysis_file, '     vectors or, if not, assumed to be needed.');
    writeln( analysis_file );
  END; { if }
END; { while }
END.
```

C.3. Microprocessor Description File, MICRO_FILE

The following enclosure is a copy of MICRO_FILE which describes the defined instructions within the Motorola 68000 microprocessor instruction set. Similar versions of MICRO_FILE can be designed for the Motorola 68010 and 68020 microprocessor instruction sets.

The first line of the file contains the number of remaining lines in the file, in this case 359. Each of the remaining lines of this file contains a 16 character code which may or may nc be followed by a comment string. For instance, consider the second line of the MICRO_FILE example,



16 character code comment string

The 16 character code represents a 16-bit opcode value, where the most significant bit is leftmost (or the first character read). Characters '1', '0', and 'X' denote logic values '1', '0', and 'don't care' (i.e. either logic value) respectively. The arbitrary logic representation allows multiple opcode values to be represented by a single entry in MICRO_FILE. This is particularly beneficial in the case of the Motorola 68000 microprocessor instruction set which has 43342 defined instructions. Comment strings following character codes describe the instruction represented. Some instructions require several character codes to describe their opcode values, in which case a comment string is only attached to the first entry associated with that instruction.

```
359
1100XXX10000XXXX ABCD
1101XXXXX0XX0XXX ADD
1101XXXXX0001XXX
1101XXXXX0011XXX
1101XXXXX0101XXX
1101XXXXX01110XX
1101XXXXX0111100
1101XXXX01XX0XXX
1101XXXX01001XXX
1101XXXX01011XXX
1101XXXX01101XXX
1101XXXX011110XX
1101XXXX01111100
1101XXXX110XXXXX ADDA
1101XXXX1110XXXX
1101XXXX11110XXX
1101XXXX111110XX
1101XXXX11111100
00000110OXXX0XXX ADDI
000001100X011XXX
000001100X101XXX
000001100X11100X
0000011010XX0XXX
0000011010011XXX
0000011010101XXX
000001101011100X
0101XXX00X0XXXXX ADDQ
0101XXX00X10XXXX
0101XXX00X110XXX
0101XXX00X11100X
0101XXX0100XXXXX
0101XXX01010XXXX
0101XXX010110XXX
0101XXX01011100X
1101XXX10X00XXXX ADDX
1101XXX11000XXXX
1100XXXXX0XX0XXX AND
1100XXXXX0011XXX
1100XXXXX0101XXX
1100XXXXX01110XX
1100XXXXX0111100
1100XXXX01XX0XXX
1100XXXX01011XXX
1100XXXX01101XXX
1100XXXX011110XX
1100XXXX01111100
00000010OXXX0XXX ANDI
000000100X011XXX
000000100X101XXX
000000100X11100X
0000001010XX0XXX
0000001010011XXX
0000001010101XXX
000000101011100X
000000100X111100 ANDI TO CCR , ANDI TO SR
1110XXXX0XX00XXX ASL , ASR
1110XXXX10X00XXX
1110000X1101XXXX
1110000X1110XXXX
1110000X11110XXX
1110000X1111100X
0110XXXXXXXXXXXX Bcc
0000XXX101XX0XXX BCHG
0000XXX101011XXX
0000XXX101101XXX
0000XXX101110XXX
0000XXX10111100X
0000100001XX0XXX
0000100001011XXX
0000100001101XXX
0000100001110XXX
000010000111100X
0000XXX110XX0XXX BCLR
0000XXX110011XXX
0000XXX110101XXX
0000XXX11011100X
0000100010XX0XXX
0000100010011XXX
0000100010101XXX
000010001011100X
0110000OXXXXXXXX BRA
0000XXX111XX0XXX BSET
0000XXX111011XXX
0000XXX111101XXX
0000XXX11011100X
0000100011XX0XXX
0000100011011XXX
0000100011101XXX
000010001111100X
0110000lXXXXXXXX BSR
0000XXX100XX0XXX BTST
0000XXX100011XXX
0000XXX100101XXX
0000XXX1001110XX
0000XXX100111100
0000100000XX0XXX
0000100000011XXX
0000100000101XXX
0000100000111OXX
0100XXX110XX0XXX CHK
0100XXX110011XXX
0100XXX110101XXX
0100XXX1101110XX
0100XXX110111100
0100001OOXXX0XXX CLR
010000100X011XXX
010000100X101XXX
010000100X11100X
0100001010XX0XXX
0100001010101XXX
0100001010011XXX
0100001010111OOX
1011XXX00X0XXXXX CMP
1011XXX00X10XXXX
1011XXX00X110XXX
1011XXX00X1110XX
1011XXX00X111100
1011XXX0100XXXXX
1011XXX01010XXXX
```

```
1011XXX010110XXX
1011XXX010111XXX
1011XXX010111100
1011XXXX110XXXXX CMPA
1011XXXX1110XXXX
1011XXXX11110XXX
1011XXXX111110XX
1011XXXX111111100
000011000XXX0XXX CMPI
000011000X011XXX
000011000X101XXX
000011000X11100X
000011001OXX0XXX
000011001001IXXX
000011001010IXXX
000011001011100X
1011XXX10X001XXX CMPM
1011XXX110001XXX
0101XXXX11001XXX BDcc
1000XXXX11XX0XXX DIVS , DIVU
1000XXXX11011XXX
1000XXXX11101XXX
1000XXXX111110XX
1000XXXX11111100
1011XXX10XXX0XXX EOR
1011XXX10X011XXX
1011XXX10X101XXX
1011XXX10X11100X
1011XXX110XX0XXX
1011XXX110011XXX
1011XXX110101XXX
1011XXX11011100X
000010100XXX0XXX EORI
000010100X011XXX
000010100X101XXX
000010100X11100X
000010101OXX0XXX
000010101001IXXX
000010101010IXXX
000010101011100X
000010100X111100 EORI TO CCR , EORI TO SR
1100XXX10100XXXX EXG
1100XXX110001XXX
010010001X000XXX EXT
0100101011111100 ILLEGAL
010011101XX10XXX JMP , JSR
010011101X101XXX
010011101X1110XX
0100XXX111X10XXX LEA
0100XXX111101XXX
0100XXX1111110XX
010011100101OXXX LINK
1110XXXX0XX0XXXX LSL , LSR , ROL , ROR
1110XXXX10X0XXXX
11100X1X1101XXXX
11100X1X1110XXXX
11100X1X11110XXX
11100X1X1111100X
0001XXX0000XXXXX MOVE (BYTE)
0001XXX00010XXXX
0001XXX000110XXX
0001XXX0001110XX
0001XXX000111100
0001XXX01X0XXXXX
0001XXX01X10XXXX
0001XXX01X110XXX
0001XXX01X1110XX
0001XXX01X111100
0001XXX10X0XXXXX
0001XXX10X10XXXX
0001XXX10X110XXX
0001XXX10X1110XX
0001XXX10X111100
0001XXX1100XXXXX
0001XXX11010XXXX
0001XXX110110XXX
0001XXX1101110XX
0001XXX110111100
000100011110XXXX
000100011110XXXX
000100011111OXXX
000100011111110X
0001000111111100
000100111110XXXX
000100111110XXXX
000100111111OXXX
000100111111110X
0001001111111100
001XXXX0000XXXXX MOVE (WORD & LONG)
001XXXX00010XXXX
001XXXX000110XXX
001XXXX0001110XX
001XXXX000111100
001XXXX01X0XXXXX
001XXXX01X10XXXX
001XXXX01X110XXX
001XXXX01X1110XX
001XXXX01X111100
001XXXX10X0XXXXX
001XXXX10X10XXXX
001XXXX10X110XXX
001XXXX10X1110XX
001XXXX10X111100
001XXXX1100XXXXX
001XXXX11010XXXX
001XXXX110110XXX
001XXXX1101110XX
001XXXX110111100
001X0001110XXXXX
001X000111110XXXX
001X000111111OXXX
001X0001111110XX
001X000111111100
001X0011110XXXXX
001X00111110XXXX
001X00111111OXXX
001X001111111OXX
001X0011111111100
010001X011XX0XXX MOVE TO CCR , MOVE TO SR (P)
010001X011011XXX
```

```
010001X011101XXX
010001X0111110XX
010001X011111100
0100000011XX0XXX MOVE FROM SR
0100000011011XXX
0100000011101XXX
0100000011111100
0100111001102XXX MOVE USP (P)
001XXXX0010XXXXX MOVEA
001XXXX00110XXXX
001XXXX001110XXX
001XXXX0011110XX
001XXXX001111100
01001X001X01XXX MOVEM
01001X001X10XXXX
01001X001X110XXX
01001X001X1110XX
0000XXX1XX001XXX MOVEP
01111110XXXXXXXX MOVEQ
1100XXXX11XX0XXX MULS , MULU
1100XXXX11011XXX
1100XXXX11101XXX
1100XXXX111110XX
1100XXXX11111100
0100100000XX0XXX NBCD
0100100000011XXX
0100100000101XXX
0100100000111100
01000X000XXX0XXX NEG , NEGX
01000X000X011XXX
01000X000X101XXX
01000X000X11100X
01000X0010XX0XXX
01000X0010011XXX
01000X0010101XXX
01000X001011100X
0100111001110001 NOP
0100011100XX0XXX NOT
0100011100X011XXX
0100011100X101XXX
0100011100X11100X
0100011010XX0XXX
0100011010011XXX
0100011010101XXX
0100011010111100X
1000XXXXX0XX0XXX OR
1000XXXXX0011XXX
1000XXXXX0101XXX
1000XXXXX01110XX
1000XXXXX0111100
1000XXXX01XX0XXX
1000XXXX01011XXX
1000XXXX01101XXX
1000XXXX011110XX
1000XXXX01111100
000000000XXX0XXX ORI
000000000X011XXX
000000000X101XXX
000000000X11100X
0000000010XX0XXX
0000000010011XXX
0000000010101XXX
0000000010111100X
000000000X111100 ORI TO CCR , ORI TO SR
0100100001X10XXX PEA
0100100001101XXX
0100100001111XXX
0100111001110000 RESET (P)
1110XXXX0XX10XXX ROXL , ROXR
1110XXXX0X100XXX
1110010X1101XXXX
1110010X1110XXXX
1110010X11110XXX
1110010X1111100X
0100111001110O1X RTE , STOP
0100111001110X1 RTR , RTS
1000XXX10000XXXX SBCD
0101XXXX11XX0XXX Scc
0101XXXX11011XXX
0101XXXX11101XXX
0101XXXX1111100X
1001XXXXXX0XXXXX SUB , SUBA
1001XXXXXX10XXXX
1001XXXXXX110XXX
1001XXXXXX1110XX
1001XXXXXX111100
000001000X0XXXXX SUBI
000001000X011XXX
000001000X101XXX
000001000X11100X
0000010010010XXXX
0000010010011XXX
0000010010101XXX
000001001011100X
0101XXX10X0XXXXX SUBQ
0101XXX10X10XXXX
0101XXX10X110XXX
0101XXX10X11100X
0101XXX1100XXXXX
0101XXX11010XXXX
0101XXX110110XXX
0101XXX11011100X
1001XXX10X00XXXX SUBX
1001XXX11000XXXX
0100100001000XXX SWAP
0100101011XX0XXX TAS
0100101011011XXX
0100101011101XXX
0100101011111100X
0100111001001XXXX TRAP
0100111001110110 TRAPV
0100101001XX0XXX TST
0100101001011XXX
0100101001101XXX
0100101001111100X
0100101010XX0XXX
0100101010011XXX
0100101010101XXX
0100101010111100X
0100111001011XXX UNLK
```

C.4. Target Software, CODE_FILE

As described in Chapter 6, CODE_FILE contains the target software to be processed by PARUT. The software is presented to PARUT in a machine code representation. The example of CODE_FILE in this enclosure is generated by the UNIX 'adb' facility.

CODE_FILE can be split into three sections. The first three lines of the file contain redundant information which is ignored when the file is processed by PARUT. The remaining two sections are each generated by an 'adb' command of the form,

$$< address >, < count >< request >< modifier >$$

where $< address >$ specifies the location from which processing commences, $< count >$ specifies the number of consecutive locations to be processed, and $< request >$ specifies the output of the operation specified by $< modifier >$. Further details of the UNIX 'adb' facility can be found in Bourne [1982].

The next section of CODE_FILE contains a source code dump and is produced by the command,

lstart,180?i

The second section contains information on the location of opcodes within the source code and is generated by the command,

lstart,180?x

In both 'adb' commands, 'lstart' is a label in the source code denoting the start of the information to be retrieved, '180' is the hexadecimal number of code lines to be extracted, '?' specifies output to the file system file 'a.out' (later renamed CODE_FILE), and 'i' and 'x' specify code dump and opcode information respectively.

HEADER

```
a.out file = a.out
cannot open core
ready
```

SECTION 2

```
lstart:  0x13FC 0xFF 0x2 0x6602 0x13FC 0x0 0x2 0x6604
  0x13FC 0x92 0x2 0x660E 0x13FC 0x1 0x2 0x660A
  0x13FC 0xFF 0x2 0x660C 0x203C 0x0 0x1 0x41F9
  0x0 0x7A 0x4E4F 0xF 0x223C 0x0 0x0 0x1239 0x2
  0x6608 0x201 0x1 0x6600 0x10 0x13FC 0x4 0x2
  0x6606 0x4EF9 0x0 0x60
ldrain:  0x13FC 0x1 0x2 0x6606
loop:  0x27C 0xF8FF 0x67F9 0x0 0x0 0x4E4F 0xA 0x4E71 0x4E71
  0x4E71 0x4E71 0x4EF9 0x0 0x60
lint:  0x2F01 0x223C 0x0 0x0 0x1239 0x2 0x660C 0x201
  0x10 0x6700 0x3A 0x1239 0x2 0x6608 0x201 0x2
  0xC01 0x0 0x6600 0x28 0x4EB9 0x0 0x11A 0x1239
  0x2 0x6608 0x201 0x2 0xC01 0x0 0x6600 0x10
  0x13FC 0x4 0x2 0x6606 0x4EF9 0x0 0x10E
lint1:  0x223C 0x0 0x0 0x1239 0x2 0x660C 0x201 0x2
  0x6700 0x34 0x1239 0x2 0x6608 0x201 0x1 0xC01
  0x1 0x6600 0x22 0x4EB9 0x0 0x11A 0x1239 0x2
  0x6608 0x201 0x1 0xC01 0x1 0x6600 0xA 0x13FC
  0x1 0x2 0x6606
lout:  0x13FC 0xFF 0x2 0x660C 0x221F 0x4E73
lwait:  0x2F06 0x2C3C 0x0 0xFFFF
lw2:  0x486 0x0 0x1 0x6600 0xFFF8 0x2C1F 0x4E75
mess:
```

SECTION 3

```
text address not found
lstart:   mov.b &0xFF,ddra
lstart+0x8:  mov.b &0x0,ddrb
lstart+0x10:  mov.b &0x92,ier
lstart+0x18:  mov.b &0x1,pcr
lstart+0x20:  mov.b &0xFF,ifr
lstart+0x28:  mov.l &0x1,%d0
lstart+0x2E:  lea lint,%a0
lstart+0x34:  trap &0xF
lstart+0x38:  mov.l &0x0,%d1
lstart+0x3E:  mov.b irb,%d1
lstart+0x44:  and.b &0x1,%d1
lstart+0x48:  bne ldrain
lstart+0x4C:  mov.b &0x4,ora
lstart+0x54:  jmp loop
ldrain:
ldrain:   mov.b &0x1,ora
loop:
loop:   and.w &0xF8FF,%sr
loop+0x4:  lea lstart,%a3
loop+0xA:  trap &0xF
loop+0xE:  nop
loop+0x10:  nop
loop+0x12:  nop
loop+0x14:  nop
loop+0x16:  jmp loop
```

SECTION 3 (cont.)

```
lint:
lint:   mov.l %d1,-(%a7)
lint+0x2:  mov.l &0x0,%d1
lint+0x8:  mov.b ifr,%d1
lint+0xE:  and.b &0x10,%d1
lint+0x12:  beq lint1
lint+0x16:  mov.b irb,%d1
lint+0x1C:  and.b &0x2,%d1
lint+0x20:  cmp.b %d1,&0x0
lint+0x24:  bne lint1
lint+0x28:  jsr lwait
lint+0x2E:  mov.b irb,%d1
lint+0x34:  and.b &0x2,%d1
lint+0x38:  cmp.b %d1,&0x0
lint+0x3C:  bne lint1
lint+0x40:  mov.b &0x4,ora
lint+0x48:  jmp lout
lint1:
lint1:   mov.l &0x0,%d1
lint1+0x6:  mov.b ifr,%d1
lint1+0xC:  and.b &0x2,%d1
lint1+0x10:  beq lout
lint1+0x14:  mov.b irb,%d1
lint1+0x1A:  and.b &0x1,%d1
lint1+0x1E:  cmp.b %d1,&0x1
lint1+0x22:  bne lout
lint1+0x26:  jsr lwait
lint1+0x2C:  mov.b irb,%d1
lint1+0x32:  and.b &0x1,%d1
lint1+0x36:  cmp.b %d1,&0x1
lint1+0x3A:  bne lout
lint1+0x3E:  mov.b &0x1,ora
lout:
lout:   mov.b &0xFF,ifr
lout+0x8:  mov.l (%a7)+,%d1
lout+0xA:  rte
lwait:
lwait:   mov.l %d6,-(%a7)
lwait+0x2:  mov.l &0xFFFF,%d6
lw2:
lw2:   sub.l &0x1,%d6
lw2+0x6:  bne lw2
lw2+0xA:  mov.l (%a7)+,%d6
lw2+0xC:  rts
mess:
mess:
text address not found
```

C.5. Target Software with Fault Tolerance, RESULT_FILE

The application of software implemented fault tolerance can be complex. This enclosure contains an example of the enhanced code generated by PARUT when it applies fault tolerance to the target software. The format of the file is tailored to the development needs of PARUT.

RESULT_FILE contains a listing representing the target machine code which has four columns of information. The first column denotes the decimal address of an opcode or operand. The second and third columns denote whether the location content is an opcode ('true') or an operand ('false'), and the decimal value of that content respectively. The fourth column, by default, is set to 'true'. Positions marked 'false' describe identified potential invalid branches that require resolving by the insertion of a detection mechanism. Finally, the fifth column contains the relative target destination displacement of jump related instructions which have a valid or invalid interpretation. This column may also contain entries marked 'exception' which describe the action of a seed within a placed detection mechanism.

Within the listing of RESULT_FILE, the locations of detection mechanisms inserted within the target software are highlighted by a surrounding box, and potential invalid branches have arrows drawn to emphasis their location and action.

DETECTION MECHANISM PLACEMENT CODE LISTING

22000	True	5116	True	
22002	False	255	True	
22004	False	2	True	
22006	False	26114	True	4
22008	True	24578	True	4
22010	False	24577	True	Exception
22012	True	5116	True	
22014	False	0	True	
22016	False	2	True	
22018	False	26116	True	6
22020	True	24580	True	6
22022	False	24577	True	Exception
22024	False	24577	True	Exception
22026	True	5116	True	
22028	False	146	True	
22030	False	2	True	
22032	False	26126	True	16
22034	True	5116	True	
22036	False	1	True	
22038	False	2	True	
22040	False	26122	True	12
22042	True	24586	True	12
22044	False	24577	True	Exception
22046	False	24577	True	Exception
22048	False	24577	True	Exception
22050	False	24577	True	Exception
22052	False	24577	True	Exception
22054	True	5116	True	
22056	False	255	True	
22058	False	2	True	
22060	False	26124	True	14
22062	True	8252	True	
22064	False	0	True	
22066	False	1	True	
22068	True	24582	True	8
22070	False	24577	True	Exception
22072	False	24577	True	Exception
22074	False	24577	True	Exception
22076	True	16889	True	
22078	False	0	True	
22080	False	122	True	
22082	True	20047	True	
22084	False	15	True	
22086	True	8764	True	
22088	False	0	True	
22090	False	0	True	
22092	True	4665	True	
22094	False	2	True	
22096	False	26120	True	10
22098	True	513	True	
22100	False	1	True	
22102	True	24580	True	6
22104	False	24577	True	Exception
22106	False	24577	True	Exception
22108	True	26112	True	18
22110	False	16	True	
22112	True	5116	True	
22114	False	4	True	
22116	False	2	True	
22118	False	26118	True	8
22120	True	24582	True	8
22122	False	24577	True	Exception
22124	False	24577	True	Exception
22126	False	24577	True	Exception
22128	True	20217	True	-20
22130	False	0	True	
22132	False	22108	True	
22134	True	5116	True	
22136	False	1	True	
22138	False	2	True	
22140	False	26118	True	8
22142	True	636	True	
22144	False	63703	True	
22146	True	24578	True	4
22148	False	24577	True	Exception
22150	True	18425	True	
22152	False	0	True	
22154	False	0	True	
22156	True	20047	True	
22158	False	10	True	
22160	True	20081	True	
22162	True	20081	True	
22164	True	20081	True	
22166	True	20081	True	
22168	True	20217	True	-60
22170	False	0	True	
22172	False	22108	True	
22174	True	12033	True	
22176	True	8764	True	
22178	False	0	True	
22180	False	0	True	
22182	True	4665	True	
22184	False	2	True	
22186	False	26124	True	14
22188	True	513	True	
22190	False	16	True	
22192	True	26368	True	60
22194	False	58	True	
22196	True	24580	True	6
22198	False	24577	True	Exception
22200	False	24577	True	Exception
22202	True	4665	True	
22204	False	2	True	
22206	False	26120	True	10
22208	True	513	True	
22210	False	2	True	
22212	True	24580	True	6
22214	False	24577	True	Exception
22216	False	24577	True	Exception
22218	True	3073	True	
22220	False	0	True	
22222	True	26112	True	42
22224	False	40	True	
22226	True	20153	True	68
22228	False	0	True	

22230	False	22294	True	
22232	True	4665	True	
22234	False	2	True	
22236	False	26120	True	10
22238	True	513	True	
22240	False	2	True	
22242	True	24580	True	6
22244	False	24577	True	Exception
22246	False	24577	True	Exception
22248	True	3073	True	
22250	False	0	True	
22252	True	26112	True	18
22254	False	16	True	
22256	True	5116	True	
22258	False	4	True	
22260	False	2	True	
22262	False	26118	True	8
22264	True	24582	True	8
22266	False	24577	True	Exception
22268	False	24577	True	Exception
22270	False	24577	True	Exception
22272	True	20217	True	10
22274	False	0	True	
22276	False	22282	True	
22278	True	8764	True	
22280	False	0	True	
22282	False	0	True	
22284	True	4665	True	
22286	False	2	True	
22288	False	26124	True	14
22290	True	513	True	
22292	False	2	True	
22294	True	26368	True	54
22296	False	52	True	
22298	True	24580	True	6
22300	False	24577	True	Exception
22302	False	24577	True	Exception
22304	True	4665	True	
22306	False	2	True	
22308	False	26120	True	10
22310	True	513	True	
22312	False	1	True	
22314	True	24580	True	6
22316	False	24577	True	Exception
22318	False	24577	True	Exception
22320	True	3073	True	
22322	False	1	True	
22324	True	26112	True	36
22326	False	34	True	
22328	True	20153	True	-34
22330	False	0	True	
22332	False	22294	True	
22334	True	4665	True	
22336	False	2	True	
22338	False	26120	True	10
22340	True	513	True	
22342	False	1	True	
22344	True	24580	True	6
22346	False	24577	True	Exception
22348	False	24577	True	Exception

22350	True	3073	True	
22352	False	1	True	
22354	True	26112	True	12
22356	False	10	True	
22358	True	5116	True	
22360	False	1	True	
22362	False	2	True	
22364	False	26118	True	8
22366	True	24582	True	8
22368	False	24577	True	Exception
22370	False	24577	True	Exception
22372	False	24577	True	Exception
22374	True	5116	True	
22376	False	255	True	
22378	False	2	True	
22380	False	26124	True	14
22382	True	8735	True	
22384	True	20083	True	
22386	True	12038	True	
22388	True	24582	True	8
22390	False	24577	True	Exception
22392	False	24577	True	Exception
22394	False	24577	True	Exception
22396	True	11324	True	
22398	False	0	True	
22400	False	65535	True	
22402	True	1158	True	
22404	False	0	True	
22406	False	1	True	
22408	True	26112	True	-6
22410	False	65528	True	
22412	True	11295	True	
22414	True	20085	True	

C.6. PARUT Report File, ANALYSIS_FILE

This enclosure contains the tables of information generated by PARUT when it assesses the fault tolerance of target software. The first table provides a summary of hazards posed by invalid branches and, where appropriate, the inclusion of details of the fault tolerance achieved by applying a software implemented fault tolerant technique. The second table collates information regarding the distribution and action of jump related instructions during erroneous execution.

The information contained within ANALYSIS_FILE provides an indication of the target software fault tolerance. In particular, it details the recovery capability of erroneous jumps within target software. The recovery performance of the target software is dependent of the instruction sequence, a dynamic process, and hence the static analysis contained within the file has limited application. Chapter 7 describes experiments which assess the fault tolerance of target software through dynamic testing.

PROGRAM AREA : ANALYSIS

TYPE	PROGRAM AREA (BYTES)	NO. UNSEEDED OPERAND JUMPS	NO. SEED BLOCKS	AREA INCREASE
ORIGINAL CODE	308	17	0	0.000 %
DETECTION MECHANISM PLACEMENT	416	0	16	35.065 %

ANALYSIS_FILE (1)

ORIGINAL CODE ##########

	OPCODE						OPERAND					
	OUT	IN	ODD	WITHIN	NOSEED	TOTAL	OUT	IN	ODD	WITHIN	NOSEED	TOTAL
RESTART												
CHK	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DIVS/DIVU	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ILLEGAL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RESET	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TRAP	0.00	1.00	1.00	0.00	0.00	2.00	0.00	0.00	1.00	0.00	0.00	1.00
TRAPV	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
undefined	0.00	0.00	0.00	0.00	0.00	0.00	0.00	5.50	5.50	0.00	0.00	11.00
TOTAL												
UNSPECIFIED JUMP												
Bcc/BRA/BSR	0.00	7.50	0.50	0.00	0.00	8.00	0.00	17.00	0.00	0.00	17.00	17.00
JMP	0.00	2.50	0.50	0.00	0.00	3.00	0.00	0.00	0.00	0.00	0.00	0.00
JSR	0.00	2.00	0.00	0.00	0.00	2.00	0.00	0.00	0.00	0.00	0.00	0.00
TOTAL												
RETURN												
RTE	0.00	0.50	0.50	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
RTR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RTS	0.00	0.50	0.50	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
TOTAL												
STOP/WAIT												
STOP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

DETECTION MECHANISM PLACEMENT ##########

	OPCODE						OPERAND					
	OUT	IN	ODD	WITHIN	NOSEED	TOTAL	OUT	IN	ODD	WITHIN	NOSEED	TOTAL
RESTART												
CHK	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DIVS/DIVU	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ILLEGAL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RESET	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TRAP	0.00	1.00	1.00	0.00	0.00	2.00	0.00	0.00	1.00	0.00	0.00	1.00
TRAPV	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
undefined	0.00	0.00	0.00	0.00	0.00	0.00	0.00	5.50	5.50	0.00	0.00	11.00
TOTAL												
UNSPECIFIED JUMP												
Bcc/BRA/BSR	0.00	23.50	0.50	0.00	0.00	24.00	0.00	17.00	54.00	0.00	0.00	54.00
JMP	0.00	2.00	1.00	0.00	0.00	3.00	0.00	0.00	0.00	0.00	0.00	0.00
JSR	0.00	1.50	0.50	0.00	0.00	2.00	0.00	0.00	0.00	0.00	0.00	0.00
TOTAL												
RETURN												
RTE	0.00	0.50	0.50	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
RTR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RTS	0.00	0.50	0.50	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
TOTAL												
STOP/WAIT												
STOP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

ANALYSIS_FILE (2)

C.7. PARUT Diagnostics, TRACE_FILE

The final enclosure of this appendix contains an example of TRACE_FILE. This file is generated when the 'diagnostic' facility within PARUT is activated. The file contains, in chronological order, a list of procedures and functions operated by PARUT. Indented entries in the file denote nested module calls. TRACE_FILE is intended to aid analyst/programmer comprehension of the PARUT function. Examination of this file should be made in association with Chapter 6.

```
TRACE FOR PARUT
===============

program
build_adb
copy_list
build_jumps
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement next word
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
   instru_set :       True
   align_address
   instru_set :       True
   instru_set :       True
seed_list
  seed_placement (LEVEL 0)
  seed_location
    jump_direction :   22006 to    22010 : forward
    interval : clear
    test_seed : not found
    seed_details
    place_seed
  seed_location
    jump_direction :   22014 to    22020 : forward
    interval : clear
    test_seed : not found
    seed_details
    place_seed
  seed_location
    jump_direction :   22022 to    22038 : forward
    interval : clear
    test_seed : not found
    seed_details
    place_seed
  seed_location
    jump_direction :   22066 to    22076 : forward
    interval : clear
    test_seed : not found
    seed_details
    place_seed
  INFO : Pass Complete
address_list
build_jumps
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   set_jump displacement lower byte
   instru_set :       True
   set_jump displacement lower byte
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   set_jump displacement lower byte
   instru_set :       True
   set_jump displacement lower byte
   instru_set :       True
   set_jump displacement lower byte
   instru_set :       True
   instru_set :       True
```

```
instru_set :       True
instru_set :       True
set_jump displacement lower byte
   placement_deadlock :     False
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :       True
set_jump displacement lower byte
   placement_deadlock :     False
instru_set :       True
set_jump displacement lower byte
instru_set :       True
set_jump displacement lower byte
instru_set :       True
set_jump displacement lower byte
instru_set :       True
set_jump displacement lower byte
instru_set :       True
instru_set :      False
instru_set :       True
instru_set :       True
set_jump displacement lower byte
   placement_deadlock :     False
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :      False
instru_set :       True
instru_set :      False
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :       True
set_jump displacement lower byte
   placement_deadlock :     False
instru_set :       True
instru_set :       True
instru_set :       True
set_jump displacement lower byte
instru_set :       True
set_jump displacement lower byte
instru_set :       True
set_jump displacement lower byte
instru_set :       True
set_jump displacement next word
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :       True
instru_set :       True
set_jump displacement lower byte
instru_set :       True
align_address
instru_set :       True
instru_set :       True
```

```
  seed_location
    jump_direction :   22040 to    22052 : forward
    interval : clear
    test_seed : not found
    seed_details
    place_seed
  seed_location
    jump_direction :   22056 to    22070 : forward
    interval : clear
    test_seed : not found
    seed_details
    place_seed
 INFO : Pass Complete
tidy_list
address_list
align_list
     abs address
print_list
build_jumps
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :      False
   instru_set :       True
   instru_set :       True
   instru_set :       True
```

```
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
      placement_deadlock :     False
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement next word
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   instru_set :       True
   set_jump displacement lower byte
   instru_set :       True
   align_address
   instru_set :       True
   instru_set :       True
address_list
print_list
analysis : ORIGINAL CODE
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
analysis : DETECTION MECHANISM PLACEMENT
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
      placement_deadlock :     False
```

Appendix D

EXAMPLE PROGRAMS

APPENDIX D

EXAMPLE PROGRAMS

D.1. Introduction

This appendix has three main enclosures, each containing an example of the application of the software implemented fault tolerant technique proposed in this thesis. Each enclosure has a suite of program listings, each suite having a different target processor for program implementation. The first and second enclosures contain three program code listings concerning Program A and Program B respectively. The first enclosure listing details the original code written in Assembler. The second and third enclosure listing show the insertion at Assembly code level of the detection mechanisms planted by the software implemented fault tolerant technique. The second listing shows the insertion of default size detection mechanisms, and the third listing shows the insertion of an optimum size detection mechanism for each placement. The final enclosure contains Assembler code listings, like the previous enclosures, except that there is an initial high-level language listing of the original program before its compilation to Assembler code.

To aid comprehension of the listings, two types of function are marked. Firstly, erroneous jumps are highlighted by connecting the generator and destination code, which are circled, with an arrow denoting the branch direction. Secondly, inserted detection mechanisms are distinguished from the program code by their encapsulation in rectangular boxes. The detection capability of the inserted mechanisms is shown where the destination of an erroneous jump lies with a detection mechanism. Erroneous jumps whose destination is outside the memory occupied by the software are represented by arrows which terminate with a 'star' symbol denoting detection by an Access Guardian.

D.2. Program 'A' Targeting the Motorola 68000 Microprocessor

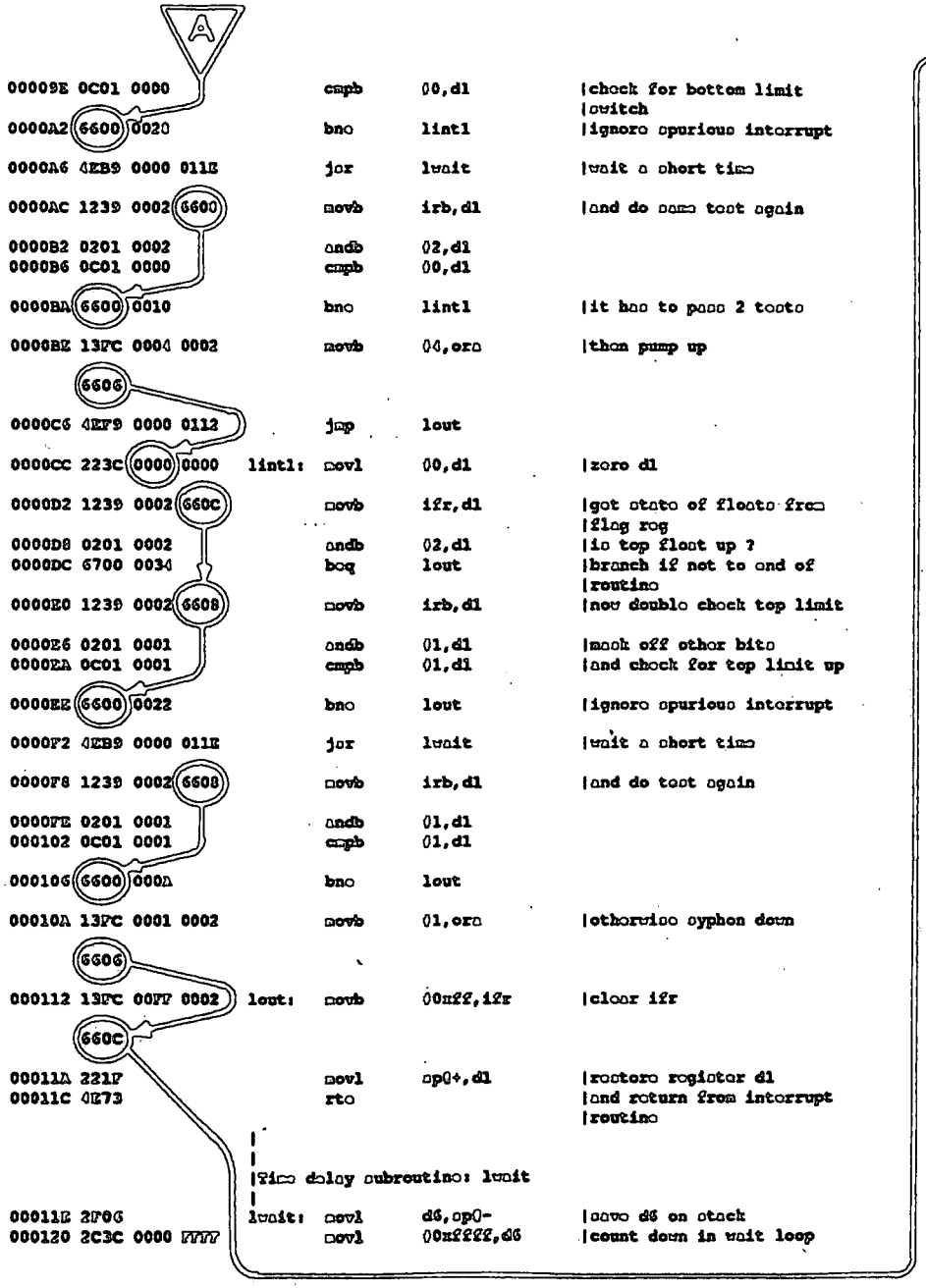
The original version of Program 'A' is written in Assembler for the 16-bit Motorola 68000 microprocessor and is shown as the first in this enclosure. The program is written as an example for the Engineering Microprocessor Laboratory at the University of Durham. It monitors two water reservoirs and controls the level of one by pumping water from or draining water to the remaining reservoir.

The second and third listings shown the insertion of detection mechanisms by the software implementation of fault tolerance proposed in this thesis. The second and third listings respectively detail default size and optimum size detection mechanism placements.

```
00000000000000000000000000000000000000000000000000
0                                                  0
0  PROGRAM REFERENCE : A                           0
0  TARGET PROCESSOR  : Motorola 68000              0
0  CODE LANGUAGE     : Assembler                   0
0                                                  0
0  VERSION 1 (ORIGINAL CODE)                       0
0                                                  0
00000000000000000000000000000000000000000000000000
```

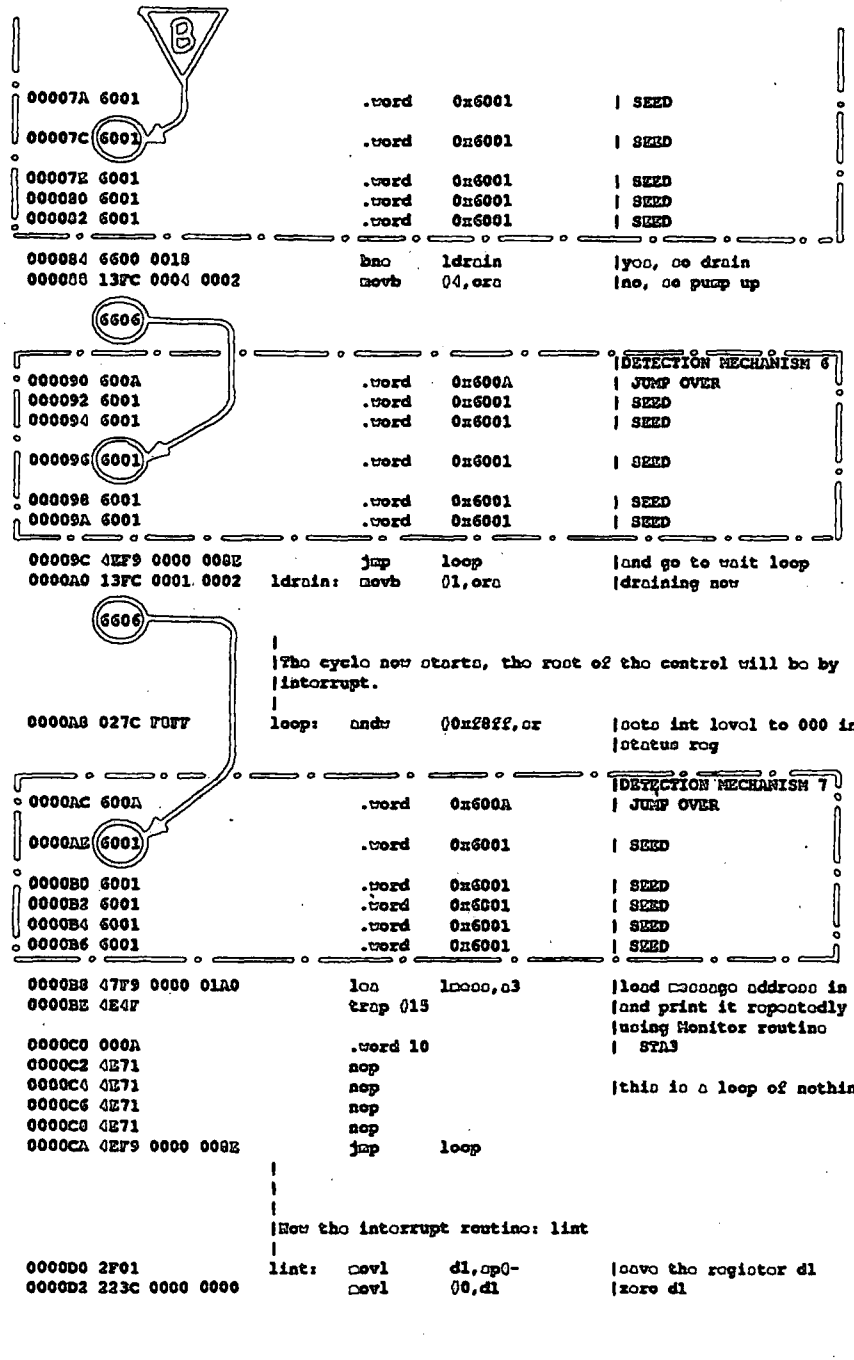
This program was originally written by Clive Prosse as an example program for the Engineering Microprocessor Laboratory. It monitors two water reservoirs and controls the water level of one by pumping from or draining water to the remaining reservoir.

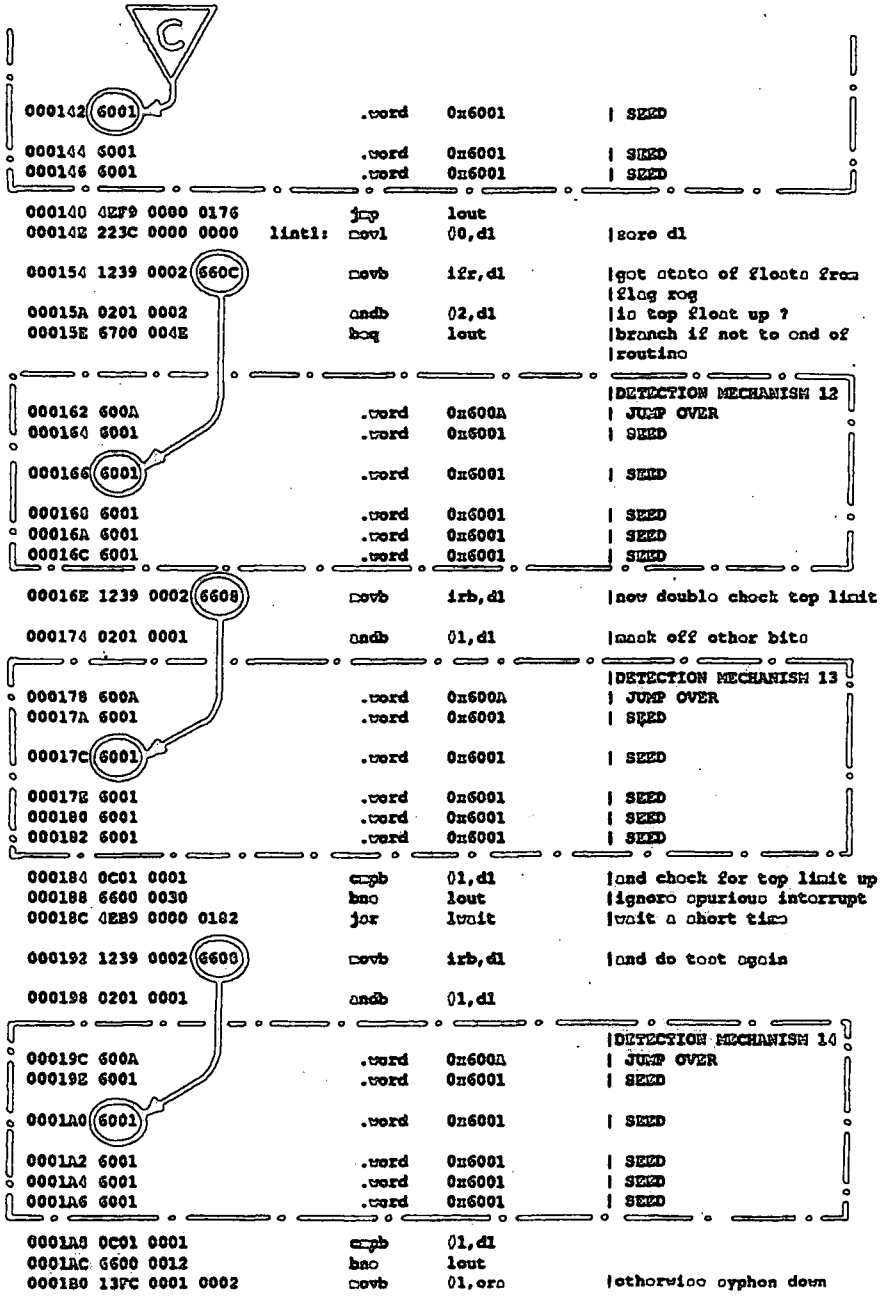
```
Addr   Code              Listing
----   ----              -------
                         |Q12(c). Pump Control using interrupts
                         |        Using registers with even byte branch formats
                         |
                         |        First set up VIA registers
       0002 6602         ddra = 0x26602  | address of data direction reg. A side
       0002 6604         ddrb = 0x26604  | address of data direction reg. B side
       0002 6606         ora  = 0x26606  | address of output reg. A side
       0002 6608         irb  = 0x26600  | address of input reg. B side
       0002 660A         pcr  = 0x2660a  | address of periph control reg
       0002 660C         ifr  = 0x2660c  | address of int flag reg
       0002 660E         ier  = 0x2660e  | address of int enable reg
                         |
000000 13FC 00FF 0002    start:  movb     #0xff,ddra      |set A side for output

000008 13FC 0000 0002            movb     #0, ddrb        |set B side for input

000010 13FC 0092 0002            movb     #0x92,ier       |enable CA1 and CB1

000018 13FC 0001 0002            movb     #1,pcr          |set active edge neg for
                                                          |CB1 bottom float

                                                          |and active edge pos for
                                                          |CA1 top float
000020 13FC 00FF 0002            movb     #0xff,ifr       |zero flag register

                         |
                         |
                         |Now we link the interrupt vector to the interrupt service
                         |routine using the Monitor commands.
                         |
000028 203C 0000 0001            movl     #1,d0           |autovector 1 specified

00002E 41F9 0000 007E            lea      lint,a0         |a0 points to interrupt
                                                          |service routine

000034 4E4F                      trap #15                 |call Monitor routine

000036 000F                      .word 15                 |  SETVEC
                         |
                         |
                         |VIA and interrupts are now set up, so pumping can begin.
                         |
                         |We must first test the state of the floats and start the
                         |process, as control will only occur when a float transition
                         |takes place.
                         |
000038 223C 0000 0000            movl     #0,d1           |clear d1

00003E 1239 0002 6608            movb     irb,d1          |check if top float is up ?

000044 0201 0001                 andb     #1,d1           |is it ?
000048 6600 0010                 bne      ldrain          |yes, so drain

00004C 13FC 0004 0002            movb     #4,ora          |no, so pump up
       6606
000054 4EF9 0000 0062            jmp      loop            |and go to wait loop

00005A 13FC 0001 0002    ldrain: movb     #1,ora          |draining now
       6606
                         |
                         |The cycle now starts, the rest of the control will be by
                         |interrupt.
                         |
000062 027C F8FF         loop:   andw     #0xf8ff,sr      |sets int level to 000 in
                                                          |status reg
000066 47F9 0000 013C            lea      lmsge,a3        |load message address in a3

00006C 4E4F                      trap #15                |and print it repeatedly
                                                          |using Monitor routine
00006E 000A                      .word 10                |  SYS3
000070 4E71                      nop
000072 4E71                      nop                     |this is a loop of nothing
000074 4E71                      nop
000076 4E71                      nop
000078 4EF9 0000 0062            jmp      loop
                         |
                         |
                         |
                         |Now the interrupt routine: lint
                         |
00007E 2F01              lint:   movl     d1,sp@-         |save the register d1
000080 223C 0000 0000            movl     #0,d1           |zero d1

000086 1239 0002 660C            movb     ifr,d1          |get state of floats using
                                                          |flag reg
00008C 0201 0010                 andb     #0x10,d1        |mask flag reg to see if
                                                          |bottom float down ?
000090 6700 003A                 beq      lint1           |branch if not

000094 1239 0002 6608            movb     irb,d1          |get limit switch state in
                                                          |d1
00009A 0201 0002                 andb     #2,d1           |mask top limit switch
```




```
.
. Object in file asm.o
.
S224020000013FC00FF0002660213FC0000002660413FC00920002660E13FC0001000266DA4D
S224020020013FC00FF0002660C203C0000000141F90002007E4E4F000F223C000000001239CB
S22402004000026608020100016600001013FC0004000266064EF9000200621BFC0001000271
S224020060660602 7CF8FF47F9000201344E4F000A4E714E714E714E714EF9000200622F01A3
S224020080223C000000001239000266DC020100106700003A1239000266080201000200C01BB
S2240200A0000066000028 4EB90002011E1239000266080201000200C010000660000101 3FC31
S2240200C0000400026606 4EF900020112223C000000001239000266DC020100026700003 40E
S2240200E01239000266080201000 10C0100016600002 24EB90002011E1239000266000201BE
S224020100001 0C0100016600000A13FC00010002660613FC00FF0002660C221F4E732F0622
S2240201202C3C0000FFFF0486000000016600FFF82C1F4E755768696C6520796F75206172F9
S224020140652077617 46368696E6720746869732049206D20636F6E74726F6C6C696E67C4
S21802016020746865207 36C757272792070756D700A0D000059
S9030000FB
```

```
00000000000000000000000000000000000000000000000000
0                                                  0
0  PROGRAM REFERENCE : A                           0
0  TARGET PROCESSOR  : Motorola 68000              0
0  CODE LANGUAGE     : Assembler                   0
0                                                  0
0  VERSION 2 (DETECTION MECHANISM PLACED)          0
0            (DEFAULT SIZE)                        0
0                                                  0
00000000000000000000000000000000000000000000000000

-----------------------------------------
This program was originally written by Clive Preece as an example program
for the Engineering Microprocessor Laboratory. It monitors two water reservoirs
and controls the water level of one by pumping from or draining water to the
remaining reservoir.
-----------------------------------------

Addr   Code             Listing
----   ----             -------
                        |Q12(c). Pump Control using interrupts with fault tolerance
                        |        Using registers with even byte branch formats
                        |
                        |        First set up VIA registers
       0002 6602        ddra = 0x26602  | address of data direction reg. A side
       0002 6604        ddrb = 0x26604  | address of data direction reg. B side
       0002 6606        ora  = 0x26606  | address of output reg. A side
       0002 6608        irb  = 0x26608  | address of input reg. B side
       0002 660A        per  = 0x2660a  | address of periph control reg
       0002 660C        ifr  = 0x2660c  | address of int flag reg
       0002 660E        ier  = 0x2660e  | address of int enable reg
                        |
000000 13FC 00FF 0002   lotart: movb    00ff,ddra       |set A side for output

       6602
000008 600A                     .word   0x600A          |DETECTION MECHANISM 1
                                                        | JUMP OVER
00000A 6001                     .word   0x6001          | SEED
00000C 6001                     .word   0x6001          | SEED
00000E 6001                     .word   0x6001          | SEED
000010 6001                     .word   0x6001          | SEED
000012 6001                     .word   0x6001          | SEED

000014 13FC 0000 0002           movb    00, ddrb        |set B side for input

       6604
00001C 600A                     .word   0x600A          |DETECTION MECHANISM 2
                                                        | JUMP OVER
00001E 6001                     .word   0x6001          | SEED
000020 6001                     .word   0x6001          | SEED
000022 6001                     .word   0x6001          | SEED
000024 6001                     .word   0x6001          | SEED
000026 6001                     .word   0x6001          | SEED

000028 13FC 0092 0002           movb    00n92,ier       |enable CA1 and CB1

       660E
000030 13FC 0001 0002           movb    01,per          |set active edge neg for
                                                        |CB1 bottom float
       660A
                                                        |and active edge pos for
                                                        |CA1 top float
                                                        |DETECTION MECHANISM 3
000038 600A                     .word   0x600A          | JUMP OVER
00003A 6001                     .word   0x6001          | SEED
00003C 6001                     .word   0x6001          | SEED
00003E 6001                     .word   0x6001          | SEED
000040 6001                     .word   0x6001          | SEED
000042 6001                     .word   0x6001          | SEED

000044 13FC 00FF 0002           movb    00ff,ifr        |zero flag register

       660C
                        |
                        |
                        |Now we link the interrupt vector to the interrupt service
                        |routine using the Monitor commands.
                        |
00004C 203C 0000 0001           movl    01,d0           |autovector 1 specified
                                                        |DETECTION MECHANISM 4
000052 600A                     .word   0x600A          | JUMP OVER
000054 6001                     .word   0x6001          | SEED
000056 6001                     .word   0x6001          | SEED
000058 6001                     .word   0x6001          | SEED
00005A 6001                     .word   0x6001          | SEED
00005C 6001                     .word   0x6001          | SEED

00005E 41F9 0000 00AE           lea     lint,a0         |a0 points to interrupt
                                                        |service routine
000064 4E4F                     trap 015                |call Monitor routine
000066 000F                     .word 15                |  SETVEC
                        |
                        |
                        |VIA and interrupts are now set up, so pumping can begin.
                        |
                        |We must first test the state of the floats and start the
                        |process, as control will only occur when a float transition
                        |takes place.
                        |
000068 223C 0000 0000           movl    00,d1           |clear d1
00006E 1239 0002 6608           movb    irb,d1          |check if top float is up ?
000074 0201 0001                andb    01,d1           |is it ?
                                                        |DETECTION MECHANISM 5
000070 600A                     .word   0x600A          | JUMP OVER
```


B
00007A 6001 .word 0x6001 | SEED
00007C 6001 .word 0x6001 | SEED
00007E 6001 .word 0x6001 | SEED
000080 6001 .word 0x6001 | SEED
000082 6001 .word 0x6001 | SEED
000084 6600 0018 bne ldrain |yes, so drain
000088 13FC 0004 0002 movb 04,ora |no, so pump up
6606
|DETECTION MECHANISM 6
000090 600A .word 0x600A | JUMP OVER
000092 6001 .word 0x6001 | SEED
000094 6001 .word 0x6001 | SEED
000096 6001 .word 0x6001 | SEED
000098 6001 .word 0x6001 | SEED
00009A 6001 .word 0x6001 | SEED
00009C 4EF9 0000 008E jmp loop |and go to wait loop
0000A0 13FC 0001 0002 ldrain: movb 01,ora |draining now
6606
|The cycle now starts, the rest of the control will be by
|interrupt.
0000A8 027C F8FF loop: andw 0#f8ff,sr |sets int level to 000 in
|status reg
|DETECTION MECHANISM 7
0000AC 600A .word 0x600A | JUMP OVER
0000AE 6001 .word 0x6001 | SEED
0000B0 6001 .word 0x6001 | SEED
0000B2 6001 .word 0x6001 | SEED
0000B4 6001 .word 0x6001 | SEED
0000B6 6001 .word 0x6001 | SEED
0000B8 47F9 0000 01A0 lea lmsg,a3 |load message address in a3
0000BE 4E4F trap 015 |and print it repeatedly
|using Monitor routine
0000C0 000A .word 10 | STA3
0000C2 4E71 nop
0000C4 4E71 nop |this is a loop of nothing
0000C6 4E71 nop
0000C8 4E71 nop
0000CA 4EF9 0000 008E jmp loop
|Now the interrupt routine: lint
0000D0 2F01 lint: movl d1,sp@- |save the register d1
0000D2 223C 0000 0000 movl 00,d1 |zero d1
0000D8 1239 0002 660C movb ifr,d1 |get state of floats using
|flag reg
0000DE 0201 0010 andb 00x10,d1 |mask flag reg to see if
|bottom float down ?
0000E2 6700 0054 beq lint1 |branch if not
|DETECTION MECHANISM 8
0000E6 600A .word 0x600A | JUMP OVER
0000E8 6001 .word 0x6001 | SEED
0000EA 6001 .word 0x6001 | SEED
0000EC 6001 .word 0x6001 | SEED
0000EE 6001 .word 0x6001 | SEED
0000F0 6001 .word 0x6001 | SEED
0000F2 1239 0002 6608 movb irb,d1 |get limit switch state in
|d1
0000F8 0201 0002 andb 02,d1 |mask top limit switch
|DETECTION MECHANISM 9
0000FC 600A .word 0x600A | JUMP OVER
0000FE 6001 .word 0x6001 | SEED
000100 6001 .word 0x6001 | SEED
000102 6001 .word 0x6001 | SEED
000104 6001 .word 0x6001 | SEED
000106 6001 .word 0x6001 | SEED
000108 0C01 0000 cmpb 00,d1 |check for bottom limit
|switch
00010C 6600 0036 bne lint1 |ignore spurious interrupt
000110 4EB9 0000 0182 jsr lwait |wait a short time
000116 1239 0002 6608 movb irb,d1 |and do same test again
00011C 0201 0002 andb 02,d1
|DETECTION MECHANISM 10
000120 600A .word 0x600A | JUMP OVER
000122 6001 .word 0x6001 | SEED
000124 6001 .word 0x6001 | SEED
000126 6001 .word 0x6001 | SEED
000128 6001 .word 0x6001 | SEED
00012A 6001 .word 0x6001 | SEED
00012C 0C01 0000 cmpb 00,d1
000130 6600 0018 bne lint1 |it has to pass 2 tests
000134 13FC 0004 0002 movb 04,ora |then pump up
6606
|DETECTION MECHANISM 11
00013C 600A .word 0x600A | JUMP OVER
00013E 6001 .word 0x6001 | SEED
000140 6001 .word 0x6001 | SEED
C

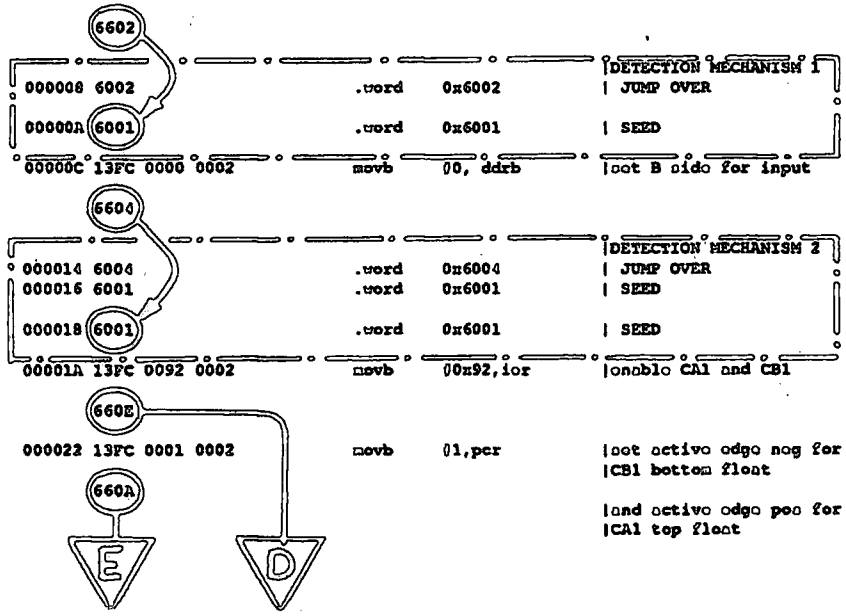

C
000142 6001 .word 0x6001 | SEED
000144 6001 .word 0x6001 | SEED
000146 6001 .word 0x6001 | SEED
000148 4EF9 0000 0176 jmp lout
00014E 223C 0000 0000 lint1: movl 0,d1 |zero d1
000154 1239 0002 660C movb ifr,d1 |get state of floats from flag reg
00015A 0201 0002 andb 2,d1 |is top float up ?
00015E 6700 004E beq lout |branch if not to end of routine
|DETECTION MECHANISM 12
000162 600A .word 0x600A | JUMP OVER
000164 6001 .word 0x6001 | SEED
000166 6001 .word 0x6001 | SEED
000168 6001 .word 0x6001 | SEED
00016A 6001 .word 0x6001 | SEED
00016C 6001 .word 0x6001 | SEED
00016E 1239 0002 6608 movb irb,d1 |now double check top limit
000174 0201 0001 andb 1,d1 |mask off other bits
|DETECTION MECHANISM 13
000178 600A .word 0x600A | JUMP OVER
00017A 6001 .word 0x6001 | SEED
00017C 6001 .word 0x6001 | SEED
00017E 6001 .word 0x6001 | SEED
000180 6001 .word 0x6001 | SEED
000182 6001 .word 0x6001 | SEED
000184 0C01 0001 cmpb 1,d1 |and check for top limit up
000188 6600 0030 bne lout |ignore spurious interrupt
00018C 4EB9 0000 0192 jsr lwait |wait a short time
000192 1239 0002 6606 movb irb,d1 |and do test again
000198 0201 0001 andb 1,d1
|DETECTION MECHANISM 14
00019C 600A .word 0x600A | JUMP OVER
00019E 6001 .word 0x6001 | SEED
0001A0 6001 .word 0x6001 | SEED
0001A2 6001 .word 0x6001 | SEED
0001A4 6001 .word 0x6001 | SEED
0001A6 6001 .word 0x6001 | SEED
0001A8 0C01 0001 cmpb 1,d1
0001AC 6600 0012 bne lout
0001B0 13FC 0001 0002 movb 1,ora |otherwise syphon down
6606
|DETECTION MECHANISM 15
0001B8 600A .word 0x600A | JUMP OVER
0001BA 6001 .word 0x6001 | SEED
0001BC 6001 .word 0x6001 | SEED
0001BE 6001 .word 0x6001 | SEED
0001C0 6001 .word 0x6001 | SEED
0001C2 6001 .word 0x6001 | SEED
0001C4 13FC 00FF 0002 lout: movb 0xff,ifr |clear ifr
660C
0001CC 221F movl sp@+,d1 |restore register d1
0001CE 4E73 rte |and return from interrupt routine
|Time delay subroutine: lwait
0001D0 2F06 lwait: movl d6,sp@- |save d6 on stack
|DETECTION MECHANISM 16
0001D2 600A .word 0x600A | JUMP OVER
0001D4 6001 .word 0x6001 | SEED
0001D6 6001 .word 0x6001 | SEED
0001D8 6001 .word 0x6001 | SEED
0001DA 6001 .word 0x6001 | SEED
0001DC 6001 .word 0x6001 | SEED
0001DE 2C3C 0000 FFFF movl 0xffff,d6 |count down in wait loop
0001E4 0406 0000 0001 lw2: subl 1,d6
0001EA 6600 FFF8 bne lw2
0001EE 2C1F movl sp@+,d6 |restore d6
0001F0 4E75 rte
0001F2 57 68 69 6C 65 lmess: .ascii "While you are watching this
I am controlling the slurry pump\n\r"
20 79 6F 75 20
61 72 65 20 77
61 74 63 68 69
6E 67 20 74 68
69 73 20 49 20
61 6D 20 63 6F
6E 74 72 6F 6C
6C 69 6E 67 20
74 68 65 20 73
6C 75 72 72 79
20 70 75 6D 70
0A 0D 00
|This program is in a continuous loop, and can only
|be stopped by pressing the reset on the MC68000.

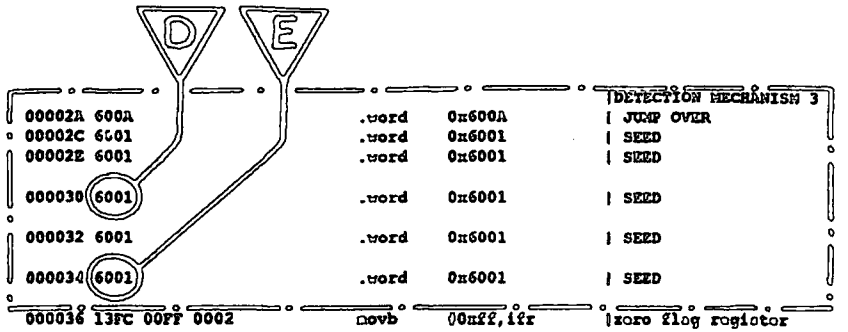
```
.
. Object in file com.o
.
S2240200013FC00FF0002660260026001130000000026604600460016001130009200025A
S224020020660E13FC00010002660A600A60016001600160016001130FC00FF0002660C203C96
S22402004000000001600660016001600141F9000200AE4E4F000F223C0000000012390002CE
S22402006066080201000160046001600166000018130FC00040002660660066001600160159
S22402008004EF90002008E13FC00010002660602700F0FF60026001470900020100A04E4F000A62
S2240200A04E714E714E714E714E714EF90002008E2F01223C0000000012390002660C0201001006
S2240200C0670000946004600160011239000266000201000260046001600100C0100006600DF
S2240200E000364EB900020102123900026600802010002600460016001000100006600018C6
S22402010013FC0004000266066006600160016001400F900020176223C0000000012390002263
S22402012066000D201000267000040E6004600160011239000266080201000160046001600181
S22402014000C0100016600003004EB900020102123900026608020100016004600160010C0176
S22402016000016600001213FC00010002660660066001600160011300FF0002660C221F33
S22402018004E732F0660066001600160012C3C0000FFFF0486000000016600FFF02C1F4E757D
S2240201A05768696C6520796F7520617265207761746368696E67207460697320492061BDC6
S2240201C020636F6E74726F6C6C696E672074686920736C7572727920707561D700A0D000022
S9030000FB
```

```
00000000000000000000000000000000000000000000000000000000
0                                                      0
0  PROGRAM REFERENCE : A                               0
0  TARGET PROCESSOR  : Motorola 68000                  0
0  CODE LANGUAGE     : Assembler                       0
0                                                      0
0  VERSION 3 (DETECTION MECHANISM PLACED)              0
0            (OPTIMUM SIZE)                            0
0                                                      0
00000000000000000000000000000000000000000000000000000000
```

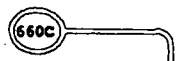
This program was originally written by Clive Prosse as an example program for the Engineering Microprocessor Laboratory. It monitors two water reservoirs and controls the water level of one by pumping from or draining water to the remaining reservoir.

```
Addr   Code             Listing
----   ----             -------
                        |Q12(c). Pump Control using interrupts with fault tolerance
                        |        Using registers with even byte branch formats
                        |
                        |        First set up VIA registers
       0002 6602        ddra = 0x26602  | address of data direction reg. A side
       0002 6604        ddrb = 0x26604  | address of data direction reg. B side
       0002 6606        ora  = 0x26606  | address of output reg. A side
       0002 6608        irb  = 0x26608  | address of input reg. B side
       0002 660A        pcr  = 0x2660a  | address of periph control reg
       0002 660C        ifr  = 0x2660c  | address of int flag reg
       0002 660E        ier  = 0x2660e  | address of int enable reg
                        |
000000 13FC 00FF 0002   initart: movb    #0xff,ddra      |set A side for output

000008 6002                      .word   0x6002          |DETECTION MECHANISM 1
00000A 6001                      .word   0x6001          | JUMP OVER
                                                         | SEED
00000C 13FC 0000 0002            movb    #0, ddrb        |set B side for input

000014 6004                      .word   0x6004          |DETECTION MECHANISM 2
000016 6001                      .word   0x6001          | JUMP OVER
                                                         | SEED
000018 6001                      .word   0x6001          | SEED

00001A 13FC 0092 0002            movb    #0x92,ier       |enable CA1 and CB1

000022 13FC 0001 0002            movb    #1,pcr          |set active edge neg for
                                                         |CB1 bottom float
                                                         |and active edge pos for
                                                         |CA1 top float
```

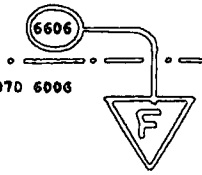





```
                                                         |DETECTION MECHANISM 3
00002A 600A                      .word   0x600A          | JUMP OVER
00002C 6001                      .word   0x6001          | SEED
00002E 6001                      .word   0x6001          | SEED

000030 6001                      .word   0x6001          | SEED

000032 6001                      .word   0x6001          | SEED

000034 6001                      .word   0x6001          | SEED

000036 13FC 00FF 0002            movb    #0xff,ifr       |zero flag register
```

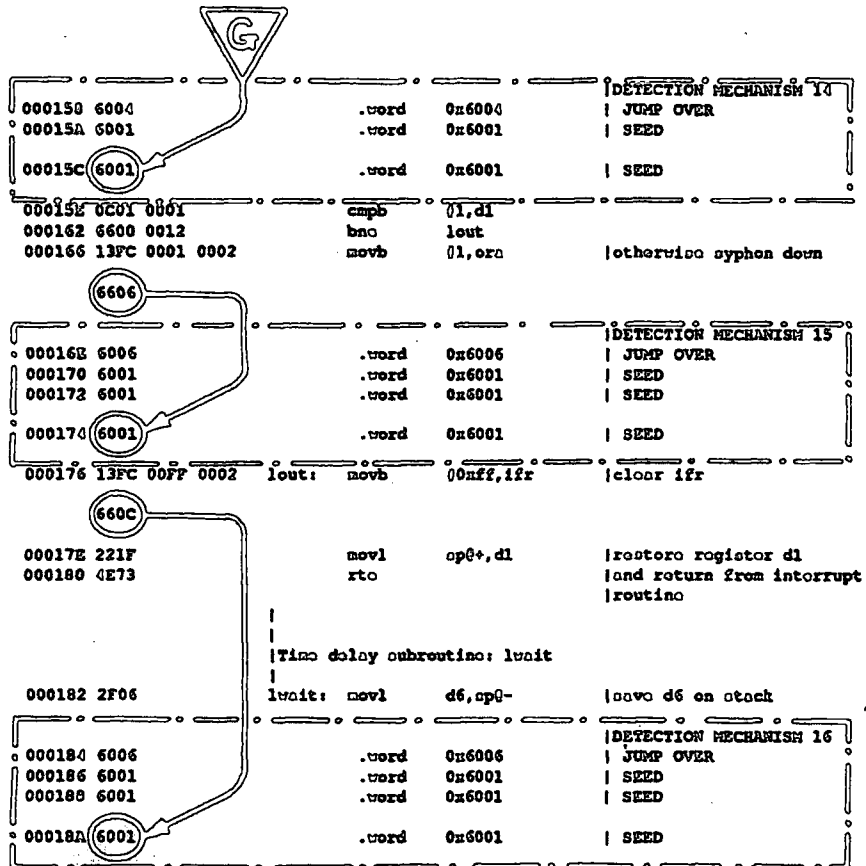



```
                        |
                        |Now we link the interrupt vector to the interrupt service
                        |routine using the Monitor commands.
                        |
00003E 203C 0000 0001            movl    #1,d0           |autovector 1 specified

                                                         |DETECTION MECHANISM 4
000044 6006                      .word   0x6006          | JUMP OVER
000046 6001                      .word   0x6001          | SEED
000048 6001                      .word   0x6001          | SEED

00004A 6001                      .word   0x6001          | SEED

00004C 41F9 0000 00AE            lea     lint,a0         |a0 points to interrupt
                                                         |service routine
000052 4E4F                      trap #15                |call Monitor routine
000054 000F                      .word 15                |  SETVEC
                        |
                        |
                        |VIA and interrupts are now set up, so pumping can begin.
                        |
                        |We must first test the state of the floats and start the
                        |process, as control will only occur when a float transition
                        |takes place.
                        |
000056 223C 0000 0000            movl    #0,d1           |clear d1

00005C 1239 0002 6608            movb    irb,d1          |check if top float is up ?

000062 0201 0001                 andb    #1,d1           |is it ?

                                                         |DETECTION MECHANISM 5
000066 6004                      .word   0x6004          | JUMP OVER
000068 6001                      .word   0x6001          | SEED

00006A 6001                      .word   0x6001          | SEED

00006C 6600 0018                 bne     ldrain          |yes, so drain
000070 13FC 0004 0002            movb    #4,ora          |no, so pump up
```




```
                                                         |DETECTION MECHANISM 6
000070 6006                      .word   0x6006          | JUMP OVER
```

```
00007A 6001                       .word    0x6001       | SEED
00007C 6001                       .word    0x6001       | SEED
00007E 6001                       .word    0x6001       | SEED
000080 4EF9 0000 008E             jmp      loop         |and go to wait loop
000086 13FC 0001 0002   ldrain:   movb     01,ors       |draining now
                        |
                        |The cycle now starts, the rest of the control will be by
                        |interrupt.
                        |
00008E 027C F8FF        loop:     andw     00xf8ff,sr   |sets int level to 000 in
                                                        |status reg
                                                        |DETECTION MECHANISM 7
000092 6002                       .word    0x6002       | JUMP OVER
000094 6001                       .word    0x6001       | SEED
000096 47F9 0000 01A0             lea      lmess,a3     |load message address in a3
00009C 4E4F                       trap 015              |and print it repeatedly
                                                        |using Monitor routine
00009E 000A                       .word 10              |  STA3
0000A0 4E71                       nop
0000A2 4E71                       nop                   |this is a loop of nothing
0000A4 4E71                       nop
0000A6 4E71                       nop
0000A8 4EF9 0000 008E             jmp      loop
                        |
                        |
                        |Now the interrupt routine: lint
                        |
0000AE 2F01             lint:     movl     d1,sp@-      |save the register d1
0000B0 223C 0000 0000             movl     00,d1        |zero d1
0000B6 1239 0002 660C             movb     ifr,d1       |get state of floats using
                                                        |flag reg
0000BC 0201 0010                  andb     00x10,d1     |mask flag reg to see if
                                                        |bottom float down ?
0000C0 6700 0054                  beq      lint1        |branch if not
                                                        |DETECTION MECHANISM 8
0000C4 6004                       .word    0x6004       | JUMP OVER
0000C6 6001                       .word    0x6001       | SEED
0000C8 6001                       .word    0x6001       | SEED
0000CA 1239 0002 6608             movb     irb,d1       |got limit switch state in
                                                        |d1
0000D0 0201 0002                  andb     02,d1        |mask top limit switch
                                                        |DETECTION MECHANISM 9
0000D4 6004                       .word    0x6004       | JUMP OVER
0000D6 6001                       .word    0x6001       | SEED
0000D8 6001                       .word    0x6001       | SEED

0000DA 0C01 0000                  cmpb     00,d1        |check for bottom limit
                                                        |switch
0000DE 6600 0036                  bne      lint1        |ignore spurious interrupt
0000E2 4EB9 0000 0182             jsr      lwait        |wait a short time
0000E8 1239 0002 6608             movb     irb,d1       |and do same test again
0000EE 0201 0002                  andb     02,d1
                                                        |DETECTION MECHANISM 10
0000F2 6004                       .word    0x6004       | JUMP OVER
0000F4 6001                       .word    0x6001       | SEED
0000F6 6001                       .word    0x6001       | SEED
0000F8 0C01 0000                  cmpb     00,d1
0000FC 6600 0018                  bne      lint1        |it has to pass 2 tests
000100 13FC 0004 0002             movb     04,ors       |then pump up
                                                        |DETECTION MECHANISM 11
000108 6006                       .word    0x6006       | JUMP OVER
00010A 6001                       .word    0x6001       | SEED
00010C 6001                       .word    0x6001       | SEED
00010E 6001                       .word    0x6001       | SEED
000110 4EF9 0000 0176             jmp      lout
000116 223C 0000 0000   lint1:    movl     00,d1        |zero d1
00011C 1239 0002 660C             movb     ifr,d1       |get state of floats from
                                                        |flag reg
000122 0201 0002                  andb     02,d1        |is top float up ?
000126 6700 004E                  beq      lout         |branch if not to end of
                                                        |routine
                                                        |DETECTION MECHANISM 12
00012A 6004                       .word    0x6004       | JUMP OVER
00012C 6001                       .word    0x6001       | SEED
00012E 6001                       .word    0x6001       | SEED
000130 1239 0002 6608             movb     irb,d1       |now double check top limit
000136 0201 0001                  andb     01,d1        |mask off other bits
                                                        |DETECTION MECHANISM 13
00013A 6004                       .word    0x6004       | JUMP OVERMECHANISM 13
00013C 6001                       .word    0x6001       | SEED
00013E 6001                       .word    0x6001       | SEED
000140 0C01 0001                  cmpb     01,d1        |and check for top limit up
000144 6600 0030                  bne      lout         |ignore spurious interrupt
000148 4EB9 0000 0182             jsr      lwait        |wait a short time
00014E 1239 0002 6608             movb     irb,d1       |and do test again
000154 0201 0001                  andb     01,d1
```



```
                                                    |DETECTION MECHANISM 14
000158 6004                     .word   0x6004      | JUMP OVER
00015A 6001                     .word   0x6001      | SEED
00015C 6001                     .word   0x6001      | SEED
00015E 0C01 0001                cmpb    #1,d1
000162 6600 0012                bne     lout
000166 13FC 0001 0002           movb    #1,ora      |otherwise syphon down
000176 13FC 00FF 0002   lout:   movb    #0xff,ifr   |clear ifr
00017E 221F                     movl    sp@+,d1     |restore register d1
000180 4E73                     rte                 |and return from interrupt
                                                    |routine
                        |
                        |Time delay subroutine: lwait
                        |
000182 2F06             lwait:  movl    d6,sp@-     |save d6 on stack
                                                    |DETECTION MECHANISM 16
000184 6006                     .word   0x6006      | JUMP OVER
000186 6001                     .word   0x6001      | SEED
000188 6001                     .word   0x6001      | SEED
00018A 6001                     .word   0x6001      | SEED
00018C 2C3C 0000 FFFF           movl    #0xffff,d6  |count down in wait loop
000192 0486 0000 0001   lw2:    subl    #1,d6
000198 6600 FFF8                bne     lw2
00019C 2C1F                     movl    sp@+,d6     |restore d6
00019E 4E75                     rts

0001A0 57 68 69 6C 65   lmsg:   .asciz "While you are watching this
                                I am controlling the slurry pump\n\r"
       20 79 6F 75 20
       61 72 65 20 77
       61 74 63 68 69
       6E 67 20 74 68
       69 73 20 49 20
       61 6D 20 63 6F
       6E 74 72 6F 6C
       6C 69 6E 67 20
       74 68 65 20 73
       6C 75 72 72 79
       20 70 75 6D 70
       0A 0D 00
```

```
                        |
                        |This program is in a continuous loop, and can only
                        |be stopped by pressing the reset on the M68000.
                        |
                        |

.
. Object in file asm.o
.
S224020000013FC00FF0002660260026001 13FC0000000266046004600160011 3FC009200025A
S224020020660E13FC00010002660A600A6001600160016001600113FC00FF0002660C203C96
S224020040000000016006600160016001 41F9000200AE4E4F000F223C0000000012390002CE
S22402006066080201000160046001600166000018 13FC0004000266066006600160016001599
S224020080 4EF90002008E13FC0001000266066027CF0FF6002600147F9000201A04E4F000A42
S2240200A04E714E714E714E714EF90002008E2F01223C0000000012390002660C0201001006
S2240200C0670000546004600160011239000266080201000260046001600 10C0100006600DF
S2240200E000364EB900020182123900026600020100026004600160010C0100006600010C6
S224020100 13FC0004000266066006600160016001 4EF900020176223C0000000012390002C3
S224020120660C0201000267000004E6004600160011239000266080201000160046001600101
S2240201400C0100016600003044EB900020182123900026608020100016004600160010C0176
S2240201600001660000121 3FC0001000266066006600160016001 13FC00FF0002660C221F35
S2240201804E732F066006600160016001 2C3C0000FFFF0406000000016600FFF82C1F4E757D
S2240201A05768696C6520796F7520617265207761746368696267207468697320492061 6DC6
S2240201C020636F6E74726F6C6C696E67207468652073 6C7572727920707 96D700A0D000022
S9030000FB
```

D.3. Program 'B' Targeting the Motorola 6805 Microprocessor

Program 'B' is written in Assembler code targeted at the 8-bit Motorola 6805 microprocessor. The first listing in this enclosure details the program code. The program demonstrates the SC687 development system for MC68(7)05 software. It simply sends a sequence of user inputs to an output device.

The insertion of detection mechanisms by the software implemented fault tolerant technique, proposed in this thesis, is shown in the second and third listings of this enclosure. Default size and optimum size detection mechanism placements are detailed in the second and third listings respectively.

```
00000000000000000000000000000000000000000000000000
0                                                  0
0  PROGRAM REFERENCE :  B                          0
0  TARGET PROCESSOR  :  Motorola 60(7)05           0
0  CODE LANGUAGE     :  Assembler                  0
0                                                  0
0  VERSION 1 (ORIGINAL CODE)                       0
0                                                  0
00000000000000000000000000000000000000000000000000
```

This program was originally written by Per Nylen (1989) to demonstrate the SC687 development system for MC68(7)05 software. It simply sends a sequence of user inputs to an output device.

```
Addr  Code      Listing
----  ----      -------

                PORTA   EQU     0               ; PORT A
                PORTB   EQU     1               ; PORT B
                PORTC   EQU     2               ; PORT C
                        ORG     $784
0784  00                DB      %10000000       ; RC-OSCILLATOR
                        ORG     $7FE
07FE  00 80             DW      $0080           ; Goto address $0080 at RESET

                        ORG     $80
0080  A6FF      START:  LDA     %11111111

0082  B7 04             STA     4               ; PORT A Output

0084  B7 05             STA     5               ; PORT B Output

0086  A6F7              LDA     %11110111

0088  B7 06             STA     6               ; PC3 Input. It is the KEY!

008A  06 02 FD  TST1:   BRSET   3,PORTC,TST1    ; Wait while key is pressed

008D  A6 DF             LDA     %11011111

008F  B7 01             STA     PORTB           ; Send code for "ONE"

0091  06 02 FD  TST2:   BRSET   3,PORTC,TST2    ; Wait while key is pressed

0094  A6 ED             LDA     %11101101

0096  B7 01             STA     PORTB           ; Send code for "TWO"

0098  06 02 FD  TST3:   BRSET   3,PORTC,TST3    ; Wait while key is pressed

009B  A6 9B             LDA     %10011011

009D  B7 01             STA     PORTB           ; Send code for "THREE"

009F  06 02 FD  TST4:   BRSET   3,PORTC,TST4    ; Wait while key is pressed

00A2  A6 B2             LDA     %10110010

00A4  B7 01             STA     PORTB           ; Send code for "FOUR"

00A6  06 02 FD  TST5:   BRSET   3,PORTC,TST5    ; Wait while key is pressed

00A9  A6 92             LDA     %10010010

00AB  B7 01             STA     PORTB           ; Send code for "FIVE"

00AD  06 02 FD  TST6:   BRSET   3,PORTC,TST6    ; Wait while key is pressed

00B0  A6 A0             LDA     %10100000

00B2  B7 01             STA     PORTB           ; Send code for "SIX"

00B4  20D4              BRA     TST1            ; Loop back
                        END
```

```
00000000000000000000000000000000000000000000000000
0                                                  0
0  PROGRAM REFERENCE :  B                          0
0  TARGET PROCESSOR  :  Motorola 68(7)05           0
0  CODE LANGUAGE     :  Assembler                  0
0                                                  0
0  VERSION 2 (DETECTION MECHANISM PLACED)          0
0            (DEFAULT SIZE)                        0
0                                                  0
00000000000000000000000000000000000000000000000000
```

This program was originally written by Per Nylen (1989) to demonstrate the SC607 development system for MC68(7)05 software. It simply sends a sequence of user inputs to an output device.

Addr	Code		Listing		
		PORTA	EQU	0	; PORT A
		PORTB	EQU	1	; PORT B
		PORTC	EQU	2	; PORT C
			ORG	$784	
0784	80		DB	%10000000	; RC-OSCILLATOR
			ORG	$7FE	
07FE	0080		DW	$0080	; Goto address $0080 at RESET
					; DETECTION MECHANISM 1
0080	20 05				; JUMP OVER
0082	21 03				; SEED
0084	83				; SEED
0085	83				; SEED
0086	83				; SEED
			ORG	$80	
0087	A6FF	START:	LDA	%11111111	
0089	B7 04		STA	4	; PORT A Output
008B	B7 05		STA	5	; PORT B Output
008D	A6F7		LDA	%11110111	
					; DETECTION MECHANISM 2
008F	20 05				; JUMP OVER
0091	21 83				; SEED
0093	83				; SEED
0094	83				; SEED
0095	83				; SEED
0096	B7 06		STA	6	; PC3 Input. It is the KEY!
0098	06 02 FD	TST1:	BRSET	3,PORTC,TST1	; Wait while key is pressed
					; DETECTION MECHANISM 3
009B	21 03				; SEED
009D	A6DF		LDA	%11011111	
009F	B7 01		STA	PORTB	; Send code for "ONE"
00A1	06 02 FD	TST2:	BRSET	3,PORTC,TST2	; Wait while key is pressed
					; DETECTION MECHANISM 4
00A4	21 83				; SEED
					; DETECTION MECHANISM 5
00A6	20 05				; JUMP OVER
00A8	21 83				; SEED
00AA	83				; SEED
00AB	83				; SEED
00AC	83				; SEED
00AD	A6ED		LDA	%11101101	
00AF	B7 01		STA	PORTB	; Send code for "TWO"
00B1	06 02 FD	TST3:	BRSET	3,PORTC,TST3	; Wait while key is pressed
					; DETECTION MECHANISM 6
00B4	21 83				; SEED
00B6	A69B		LDA	%10011011	
00B8	B7 01		STA	PORTB	; Send code for "THREE"
00BA	06 02 FD	TST4:	BRSET	3,PORTC,TST4	; Wait while key is pressed
					; DETECTION MECHANISM 7
00BD	21 83				; SEED
00BF	A6B2		LDA	%10110010	
00C1	B7 01		STA	PORTB	; Send code for "FOUR"
00C3	06 02 FD	TST5:	BRSET	3,PORTC,TST5	; Wait while key is pressed
					; DETECTION MECHANISM 8
00C6	21 83				; SEED
00C8	A692		LDA	%10010010	
00CA	B7 01		STA	PORTB	; Send code for "FIVE"
00CC	06 02 FD	TST6:	BRSET	3,PORTC,TST6	; Wait while key is pressed
					; DETECTION MECHANISM 9
00CF	21 83				; SEED
00D1	A6A0		LDA	%10100000	
00D3	B7 01		STA	PORTB	; Send code for "SIX"
00D5	20D4		BRA	TST1	; Loop back
			END		

```
00000000000000000000000000000000000000000000000000
0                                                  0
0  PROGRAM REFERENCE :  D                          0
0  TARGET PROCESSOR  :  Motorola 60(7)05           0
0  CODE LANGUAGE     :  Assembler                  0
0                                                  0
0  VERSION 3 (DETECTION MECHANISM PLACED)          0
0            (OPTIMUM SIZE)                        0
0                                                  0
00000000000000000000000000000000000000000000000000
```

This program was originally written by Per Dylon (1989) to demonstrate the SC607 development system for MC60(7)05 software. It simply sends a sequence of user inputs to an output device.

```
Addr  Code        Listing
----  ----        -------

                  PORTA     EQU     0           ; PORT A
                  PORTB     EQU     1           ; PORT B
                  PORTC     EQU     2           ; PORT C
                            ORG     $784
0784  80                    DB      %10000000   ; RC-OSCILLATOR
                            ORG     $7FE
07FE  0080                  DW      $0080       ; Goto address $0080 at RESET

0080  20 05                                     ; DETECTION MECHANISM 1
                                                ;   JUMP OVER
0082  21 83                                     ;   SEED
0084  83                                        ;   SEED

                            ORG     $80
0085  A6FF        START:    LDA     %11111111
0087  B7 04                 STA     4           ; PORT A Output
0089  B7 05                 STA     5           ; PORT B Output
008B  A6F7                  LDA     %11110111

008D  20 05                                     ; DETECTION MECHANISM 2
                                                ;   JUMP OVER
008F  21 03                                     ;   SEED
0091  B7 06                 STA     6           ; PC3 Input. It is the KEY!
0093  06 02 FD    TST1:    BRSET   3,PORTC,TST1 ; Wait while key is pressed

0096  21 83                                     ; DETECTION MECHANISM 3
                                                ;   SEED
0098  A6DF                  LDA     %11011111

009A  B7 01                 STA     PORTB       ; Send code for "ONE"

009C  06 02 FD    TST2:    BRSET   3,PORTC,TST2 ; Wait while key is pressed

009F  21 83                                     ; DETECTION MECHANISM 4
                                                ;   SEED

00A1  20 05                                     ; DETECTION MECHANISM 5
                                                ;   JUMP OVER
00A3  21 03                                     ;   SEED
00A5  83                                        ;   SEED

00A6  83                                        ;   SEED
00A7  A6ED                  LDA     %11101101

00A9  B7 01                 STA     PORTB       ; Send code for "TWO"

00AB  06 02 FD    TST3:    BRSET   3,PORTC,TST3 ; Wait while key is pressed

00AE  21 03                                     ; DETECTION MECHANISM 6
                                                ;   SEED
00B0  A69B                  LDA     %10011011

00B2  B7 01                 STA     PORTB       ; Send code for "THREE"

00B4  06 02 FD    TST4:    BRSET   3,PORTC,TST4 ; Wait while key is pressed

00B7  21 83                                     ; DETECTION MECHANISM 7
                                                ;   SEED
00B9  A6B2                  LDA     %10110010

00BB  B7 01                 STA     PORTB       ; Send code for "FOUR"

00BD  06 02 FD    TST5:    BRSET   3,PORTC,TST5 ; Wait while key is pressed

00C0  21 83                                     ; DETECTION MECHANISM 8
                                                ;   SEED
00C2  A692                  LDA     %10010010

00C4  B7 01                 STA     PORTB       ; Send code for "FIVE"

00C6  06 02 FD    TST6:    BRSET   3,PORTC,TST6 ; Wait while key is pressed

00C9  21 83                                     ; DETECTION MECHANISM 9
                                                ;   SEED
00CB  A6A0                  LDA     %10100000

00CD  B7 01                 STA     PORTB       ; Send code for "SIX"

00CF  20D4                  BRA     TST1        ; Loop back
                            END
```

D.4. Program 'C' Targeting the Intel 80386 Microprocessor

The original version of Program 'C' is written in the high-level programming language known as C. High-level language programs can be transported for application on many target processors because they describe a function in terms totally abstracted from the architectural influences of any one microprocessor system. The first listing in this enclosure details the original program. The program has no particular function: its purpose is to demonstrate the potential hazards of machine code which are transparent to the high-level language. Functions within the program use local or passed parameters.

The target processor selected for Program 'C' is the 32-bit Intel 80386. This microprocessor has been chosen so that the software implemented fault tolerant technique proposed in this thesis is demonstrated on a variety of microprocessors. In order to generate code tailored for application on the Intel 80386 processor, Program 'C' is compiled and an Assembler code representation of the source code is shown as the second listing in this enclosure.

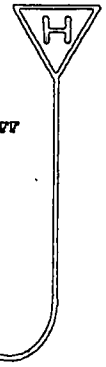
Application of the software implemented fault tolerant technique proposed in this thesis is shown within this enclosure at Assembler code level. The third and forth listing respectively detail default size and optimum size detection mechanism placements.

```
00000000000000000000000000000000000000000000
0                                            0
0  PROGRAM REFERENCE  :  C                   0
0  TARGET PROCESSOR   :  Not Specified       0
0  CODE LANGUAGE      :  C                   0
0                                            0
00000000000000000000000000000000000000000000

---------------------------------------
This program was written by Guy Wingate purely to demonstrate that high level
languages hide the potential hazards of machine code when implemented in a
microprocessor system. The program passes parameters by reference amongst
several functions.
---------------------------------------

/* function to return user input value */
int getvalue()
{ int inputvalue;
  /* get an integer value from the user at standard input */
  printf("input a value:");
  scanf("%d",&inputvalue);
  printf("inputvalue: %d\n",inputvalue);
  /* return value for function name */
  return(inputvalue);
}/* end function getvalue */

/* function to determine power of value */
int power(firstnumber, secondnumber);
int *firstnumber, *secondnumber;
{ int temptorage, i;
  for (i=0; i<=*firstnumber; ++i)
    temptorage = temptorage * (*secondnumber);
  /* return power result */
  return(temptorage);
}/* end power function */

/* function to swap values */
int swap(firstnumber, secondnumber)
int *firstnumber, *secondnumber;
{ int temptorage;
  printf("Original order: %d %d\n", *firstnumber, *secondnumber);
  temptorage = *firstnumber;
  *firstnumber = *secondnumber;
  *secondnumber = temptorage;
  printf("Swapped order: %d %d\n", *firstnumber, *secondnumber);
  /* exit function */
  return;
}/* end swap function */

/* main : get two number swap and calculate second number to power of first */
main()
{  int firstvalue, secondvalue;
   firstvalue = getvalue();
   secondvalue = getvalue();
   printf("Firstvalue : %d   Secondvalue : %d\n", firstvalue, secondvalue);
   swap(&firstvalue, &secondvalue);
   printf("Secondvalue to power of firstvalue =: %d\n",
     power(fisrtvalue, secondvalue);
   /* exit main function */
}/* end main */
```

```
00000000000000000000000000000000000000000000
0                                            0
0  PROGRAM REFERENCE :  C                    0
0  TARGET PROCESSOR  :  Intel 80386          0
0  CODE LANGUAGE     :  Assembler            0
0                                            0
0  VERSION 1  ORIGINAL CODE                  0
0                                            0
00000000000000000000000000000000000000000000
```

This is the assembler translation of a program written in C.

Addr	Code	Listing		
0000	EB33	gotvalue:	jmp	gotvalue+0x35
0002	682C210000		pushl	$otext+0x12C
0007	E8980B0000		call	otext+0x78
000C	59		popl	%ecx
000D	8D45FC		leal	-4(%ebp),%eax
0010	50		pushl	%eax
0011	683B210000		pushl	$otext+0x13B
0016	E0910B0000		call	otext+0x80
001B	83C4 08		addl	$8,%esp
001E	FF 75 FC		pushl	-4(%ebp)
0021	683E210000		pushl	$otext+0x13E
0026	E8 79 0B0000		call	otext+0x78
002B	83C408		addl	$8,%esp
002E	8B45FC		movl	-4(%ebp),%eax
0031	EB00		jmp	gotvalue+0x33
0033	C9		leave	
0034	C3		ret	
0035	55		pushl	%ebp
0036	8BEC		movl	%esp,%ebp
0038	50		pushl	%eax
0039	EBC7		jmp	gotvalue+0x2
0000	EB29	power:	jmp	power+0x2B
0002	C745F800000000		movl	$0,-8(%ebp)
0009	8B4508		movl	8(%ebp),%eax
000C	8B00		movl	(%eax),%eax
000E	3945F8		cmpl	%eax,-8(%ebp)
0011	7F11		jg	power+0x24
0013	8B450C		movl	0xC(%ebp),%eax
0016	0B55FC		movl	-4(%ebp),%edx
0019	0FAF10		imull	(%eax),%edx
001C	8955FC		movl	%edx,-4(%ebp)
001F	FF45F8		incl	-8(%ebp)
0022	EBE5		jmp	power+9
0024	8B45FC		movl	-4(%ebp),%eax
0027	EB00		jmp	power+0x29
0029	C9		leave	
002A	C3		ret	
002B	55		pushl	%ebp
002C	8BEC		movl	%esp,%ebp
002E	83EC08		subl	$8,%esp
0031	EBCF		jmp	power+2
0000	EB4C	swap:	jmp	swap+0x4E
0002	8B450C		movl	0xC(%ebp),%eax
0005	FF30		pushl	(%eax)
0007	8B4508		movl	8(%ebp),%eax
000A	FF30		pushl	(%eax)
000C	684E210000		pushl	$otext+0x14E
0011	E8200B0000		call	otext+0x70
0016	83C40C		addl	$0xC,%esp
0019	8B4508		movl	8(%ebp),%eax
001C	8B10		movl	(%eax),%edx
001E	8955FC		movl	%edx,-4(%ebp)
0021	8B4508		movl	8(%ebp),%eax
0024	8B550C		movl	0xC(%ebp),%edx
0027	8B0A		movl	(%edx),%ecx
0029	8908		movl	%ecx,(%eax)
002B	8B450C		movl	0xC(%ebp),%eax
002E	8B55FC		movl	-4(%ebp),%edx
0031	8910		movl	%edx,(%eax)
0033	8B450C		movl	0xC(%ebp),%eax
0036	FF30		pushl	(%eax)
0038	8B4508		movl	8(%ebp),%eax
003B	FF30		pushl	(%eax)
003D	6866210000		pushl	$otext+0x166
0042	E8EF0A0000		call	otext+0x70
0047	83C40C		addl	$0xC,%esp
004A	EB00		jmp	swap+0x4c
004C	C9		leave	
004D	C3		ret	
004E	55		pushl	%ebp
004F	8BEC		movl	%esp,%ebp
0051	50		pushl	%eax
0052	EBAE		jmp	swap+2
0000	EB51	main:	jmp	main+0x53
0002	E837FFFFFF		call	gotvalue
0007	8945FC		movl	%eax,-4(%ebp)
000A	E82FFF FF FF		call	gotvalue
000F	8945F8		movl	%eax,-8(%ebp)
0012	FF 75 F8		pushl	-8(%ebp)
0015	FF 75 FC		pushl	-4(%ebp)
0018	68 7D 210000		pushl	$otext+0x17D
001D	E8C00A0000		call	otext+0x70
0022	83C40C		addl	$0xC,%esp
0025	8D45F8		leal	-8(%ebp),%eax
0028	50		pushl	%eax
0029	8D45FC		leal	-4(%ebp),%eax



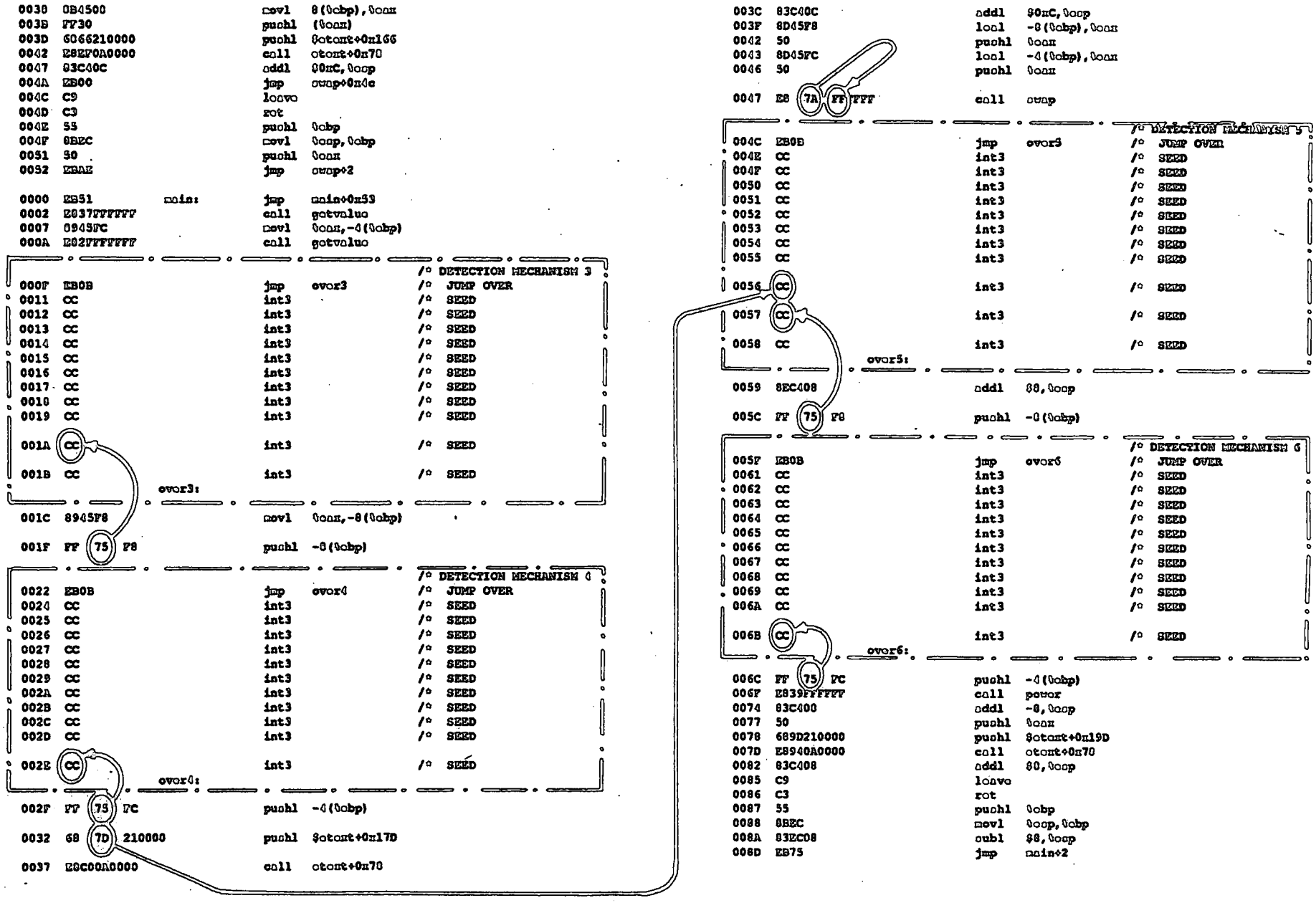
```
002C  50                 pushl  %eax
002D  E8 7A FF FF FF     call   over
0032  83C408             addl   $8,%esp
0035  FF 75 F8           pushl  -8(%ebp)
0038  FF 75 FC           pushl  -4(%ebp)
003B  E8 39 FFFFFF       call   power
0040  83C408             addl   -8,%esp
0043  50                 pushl  %eax
0044  689D210000         pushl  $start+0x19D
0049  E8940A0000         call   start+0x7E
004E  83C408             addl   $8,%esp
0051  C9                 leave
0052  C3                 ret
0053  55                 pushl  %ebp
0054  8BEC               movl   %esp,%ebp
0056  83EC00             subl   $0,%esp
0059  EBA7               jmp    main+2
```

Other routines are referenced in this code, but for the purpose of analysis the assembler program is considered to be complete.

```
000000000000000000000000000000000000000000000000
0                                              0
0  PROGRAM REFERENCE :  C                      0
0  TARGET PROCESSOR  :  Intel 80386            0
0  CODE LANGUAGE     :  Assembler              0
0                                              0
0  VERSION 2  (DETECTION MECHANISM PLACED)     0
0             (DEFAULT SIZE)                   0
0                                              0
000000000000000000000000000000000000000000000000
```

```
-------------------------------------------------
This is the assembler translation of a program written in C.
-------------------------------------------------
```

```
Addr  Code          Listing
----  ----          -------


0000  EB33          gotvalue:   jmp   gotvalue+0x35
0002  682C210000                pushl $otext+0x12C
0007  E8980B0000                call  otext+0x78
000C  59                        popl  %ecx
000D  8D45FC                    leal  -4(%ebp),%eax
0010  50                        pushl %eax
0011  683B210000                pushl $otext+0x13B
0016  E8910B0000                call  otext+0x80
001B  83C408                    addl  $8,%esp
                                                    /* DETECTION MECHANISM 1
001E  EB0B                      jmp   over1         /*  JUMP OVER
0020  CC                        int3                /*  SEED
0021  CC                        int3                /*  SEED
0022  CC                        int3                /*  SEED
0023  CC                        int3                /*  SEED
0024  CC                        int3                /*  SEED
0025  CC                        int3                /*  SEED
0026  CC                        int3                /*  SEED
0027  CC                        int3                /*  SEED
0028  CC                        int3                /*  SEED
0029  CC                        int3                /*  SEED
002A  CC                        int3                /*  SEED
              over1:
002B  FF 75 FC                  pushl -4(%ebp)
002E  683E210000                pushl $otext+0x13E
0033  E8 79 0B0000              call  otext+0x78
0038  83C408                    addl  $8,%esp
003B  8B45FC                    movl  -4(%ebp),%eax
                                                    /* DETECTION MECHANISM 2
003E  EB0B                      jmp   over2         /*  JUMP OVER
0040  CC                        int3                /*  SEED
0041  CC                        int3                /*  SEED
0042  CC                        int3                /*  SEED
0043  CC                        int3                /*  SEED
0044  CC                        int3                /*  SEED
0045  CC                        int3                /*  SEED
0046  CC                        int3                /*  SEED
0047  CC                        int3                /*  SEED
0048  CC                        int3                /*  SEED
0049  CC                        int3                /*  SEED
004A  CC                        int3                /*  SEED
              over2:
003B  EB00                      jmp   gotvalue+0x4D
004D  C9                        leave
004E  C3                        ret
004F  55                        pushl %ebp
0050  8BEC                      movl  %esp,%ebp
0052  50                        pushl %eax
0053  EBC7                      jmp   gotvalue+0x2


0000  EB29          power:      jmp   power+0x2B
0002  C745F800000000            movl  $0,-8(%ebp)
0009  8B4508                    movl  8(%ebp),%eax
000C  8B00                      movl  (%eax),%eax
000E  3945F8                    cmpl  %eax,-8(%ebp)
0011  7F11                      jg    power+0x24
0013  8B450C                    movl  0xC(%ebp),%eax
0016  8B55FC                    movl  -4(%ebp),%edx
0019  0FAF10                    imull (%eax),%edx
001C  8955FC                    movl  %edx,-4(%ebp)
001F  FF45F8                    incl  -8(%ebp)
0022  EBE5                      jmp   power+9
0024  8B45FC                    movl  -4(%ebp),%eax
0027  EB00                      jmp   power+0x29
0029  C9                        leave
002A  C3                        ret
002B  55                        pushl %ebp
002C  8BEC                      movl  %esp,%ebp
002E  83EC08                    subl  $8,%esp
0031  EBCF                      jmp   power+2


0000  EB4C          swap:       jmp   swap+0x4E
0002  8B450C                    movl  0xC(%ebp),%eax
0005  FF30                      pushl (%eax)
0007  8B4508                    movl  8(%ebp),%eax
000A  FF30                      pushl (%eax)
000C  684E210000                pushl $otext+0x14E
0011  E8200B0000                call  otext+0x78
0016  83C40C                    addl  $0xC,%esp
0019  8B4508                    movl  8(%ebp),%eax
001C  8B10                      movl  (%eax),%edx
001E  8955FC                    movl  %edx,-4(%ebp)
0021  8B4508                    movl  8(%ebp),%eax
0024  8B550C                    movl  0xC(%ebp),%edx
0027  8B0A                      movl  (%edx),%ecx
0029  8908                      movl  %ecx,(%eax)
002B  8B450C                    movl  0xC(%ebp),%eax
002E  8B55FC                    movl  -4(%ebp),%edx
0031  8910                      movl  %edx,(%eax)
0033  8B450C                    movl  0xC(%ebp),%eax
0036  FF30                      pushl (%eax)
```


0030 8B4500 movl 8(%ebp),%eax
003B FF30 pushl (%eax)
003D 6866210000 pushl $otext+0x166
0042 E8EF0A0000 call otext+0x70
0047 83C40C addl $0xC,%esp
004A EB00 jmp swap+0xde
004C C9 leave
004D C3 ret
004E 55 pushl %ebp
004F 8BEC movl %esp,%ebp
0051 50 pushl %eax
0052 EBAE jmp swap+2
0000 EB51 main: jmp main+0x53
0002 E837FFFFFF call getvalue
0007 8945FC movl %eax,-4(%ebp)
000A E82FFFFFFF call getvalue
/* DETECTION MECHANISM 3
000F EB0B jmp over3 /* JUMP OVER
0011 CC int3 /* SEED
0012 CC int3 /* SEED
0013 CC int3 /* SEED
0014 CC int3 /* SEED
0015 CC int3 /* SEED
0016 CC int3 /* SEED
0017 CC int3 /* SEED
0018 CC int3 /* SEED
0019 CC int3 /* SEED
001A CC int3 /* SEED
001B CC int3 /* SEED
over3:
001C 8945F8 movl %eax,-8(%ebp)
001F FF 75 F8 pushl -8(%ebp)
/* DETECTION MECHANISM 4
0022 EB0B jmp over4 /* JUMP OVER
0024 CC int3 /* SEED
0025 CC int3 /* SEED
0026 CC int3 /* SEED
0027 CC int3 /* SEED
0028 CC int3 /* SEED
0029 CC int3 /* SEED
002A CC int3 /* SEED
002B CC int3 /* SEED
002C CC int3 /* SEED
002D CC int3 /* SEED
002E CC int3 /* SEED
over4:
002F FF 75 FC pushl -4(%ebp)
0032 68 7D 210000 pushl $otext+0x17D
0037 E80C0A0000 call otext+0x70
003C 83C40C addl $0xC,%esp
003F 8D45F8 leal -8(%ebp),%eax
0042 50 pushl %eax
0043 8D45FC leal -4(%ebp),%eax
0046 50 pushl %eax
0047 E8 7A FFFFFF call swap
/* DETECTION MECHANISM 5
004C EB0B jmp over5 /* JUMP OVER
004E CC int3 /* SEED
004F CC int3 /* SEED
0050 CC int3 /* SEED
0051 CC int3 /* SEED
0052 CC int3 /* SEED
0053 CC int3 /* SEED
0054 CC int3 /* SEED
0055 CC int3 /* SEED
0056 CC int3 /* SEED
0057 CC int3 /* SEED
0058 CC int3 /* SEED
over5:
0059 83C408 addl $8,%esp
005C FF 75 F8 pushl -8(%ebp)
/* DETECTION MECHANISM 6
005F EB0B jmp over6 /* JUMP OVER
0061 CC int3 /* SEED
0062 CC int3 /* SEED
0063 CC int3 /* SEED
0064 CC int3 /* SEED
0065 CC int3 /* SEED
0066 CC int3 /* SEED
0067 CC int3 /* SEED
0068 CC int3 /* SEED
0069 CC int3 /* SEED
006A CC int3 /* SEED
006B CC int3 /* SEED
over6:
006C FF 75 FC pushl -4(%ebp)
006F E839FFFFFF call power
0074 83C400 addl -8,%esp
0077 50 pushl %eax
0078 689D210000 pushl $otext+0x19D
007D E8940A0000 call otext+0x70
0082 83C408 addl $8,%esp
0085 C9 leave
0086 C3 ret
0087 55 pushl %ebp
0088 8BEC movl %esp,%ebp
008A 83EC08 subl $8,%esp
008D EB75 jmp main+2

Other routines are referenced in this code, but for the purpose of analysis the assembler program is considered to be complete.

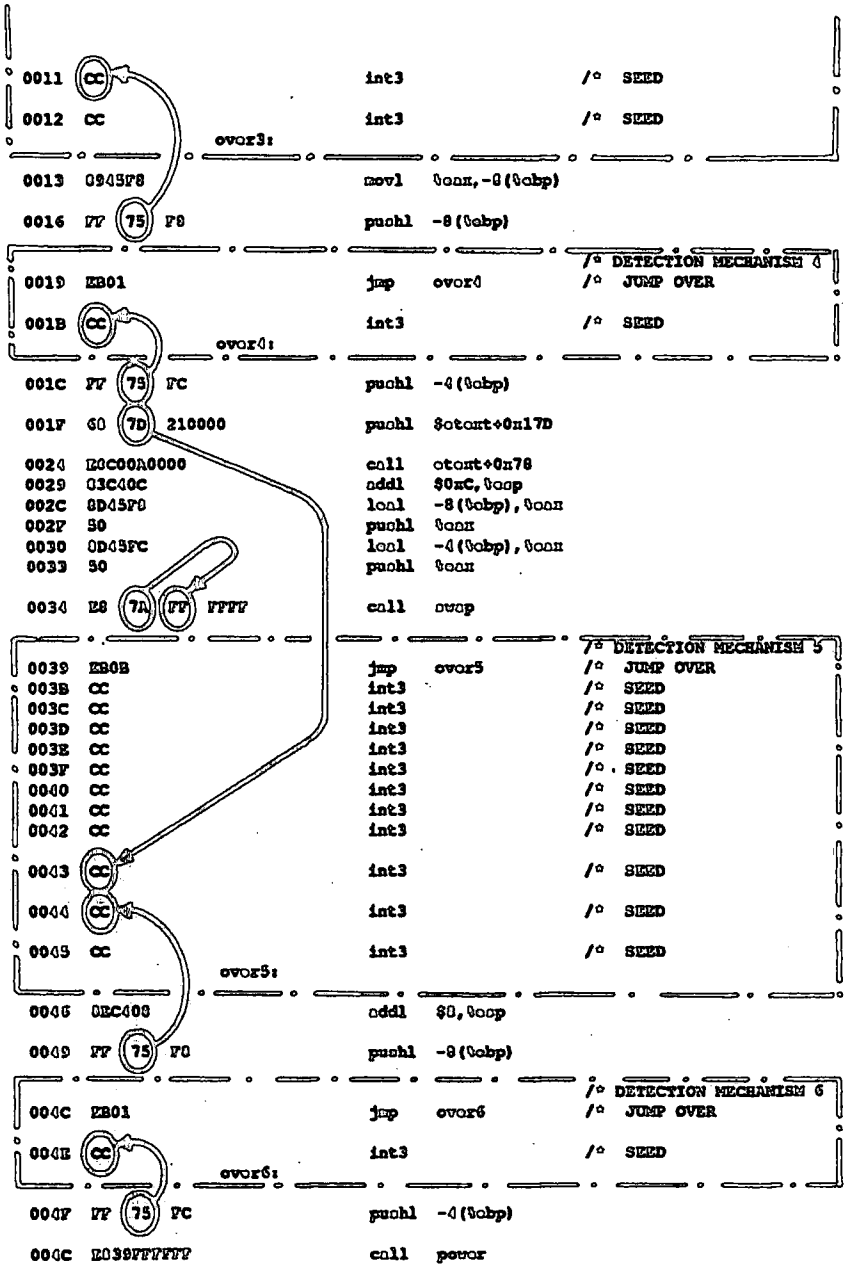
```
00000000000000000000000000000000000000000000000000
0                                                  0
0  PROGRAM REFERENCE  :  C                         0
0  TARGET PROCESSOR   :  Intel 80386               0
0  CODE LANGUAGE      :  Assembler                 0
0                                                  0
0  VERSION 3   (DETECTION MECHANISM PLACED)        0
0              (OPTIMUM SIZE)                      0
0                                                  0
00000000000000000000000000000000000000000000000000
```

This is the assembler translation of a program written in C.

```
Addr  Code          Listing
----  ----          -------

0000  EB33          gotvalue:   jmp    gotvalue+0x35
0002  682C210000                pushl  $etext+0x12C
0007  E8980B0000                call   etext+0x78
000C  59                        popl   %ecx
000D  8D45FC                    leal   -4(%ebp),%eax
0010  50                        pushl  %eax
0011  683B210000                pushl  $etext+0x13B
0016  E8910B0000                call   etext+0x80
001B  83C408                    addl   $8,%esp
                                                    /* DETECTION MECHANISM 1
001E  EB01                      jmp    over1        /*  JUMP OVER
0020  CC                        int3                /*  SEED
                    over1:
0021  FF 75 FC                  pushl  -4(%ebp)
0024  683E210000                pushl  $etext+0x13E
0029  E8 79 0B0000              call   etext+0x78
002E  83C408                    addl   $8,%esp
0031  8B45FC                    movl   -4(%ebp),%eax
                                                    /* DETECTION MECHANISM 2
0034  EB01                      jmp    over2        /*  JUMP OVER
0036  CC                        int3                /*  SEED
0037  CC                        int3                /*  SEED
                    over2:
0038  EB00                      jmp    gotvalue+0x3A
003A  C9                        leave
003B  C3                        ret
003C  55                        pushl  %ebp
003D  8BEC                      movl   %esp,%ebp
003F  50                        pushl  %eax
0040  EBC7                      jmp    gotvalue+0x2

0000  EB29          power:      jmp    power+0x2D
```



```
0002  C745F8000000000           movl   $0,-8(%ebp)
0009  8B4508                    movl   8(%ebp),%eax
000C  8B00                      movl   (%eax),%eax
000E  3945F8                    cmpl   %eax,-8(%ebp)
0011  7F11                      jg     power+0x24
0013  8B450C                    movl   0xC(%ebp),%eax
0016  8B55FC                    movl   -4(%ebp),%edx
0019  0FAF10                    imull  (%eax),%edx
001C  8955FC                    movl   %edx,-4(%ebp)
001F  FF45F8                    incl   -8(%ebp)
0022  EBE5                      jmp    power+9
0024  8B45FC                    movl   -4(%ebp),%eax
0027  EB00                      jmp    power+0x29
0029  C9                        leave
002A  C3                        ret
002B  55                        pushl  %ebp
002C  8BEC                      movl   %esp,%ebp
002E  83EC08                    subl   $8,%esp
0031  EBC7                      jmp    power+2

0000  EB4C          swap:       jmp    swap+0x4E
0002  8B450C                    movl   0xC(%ebp),%eax
0005  FF30                      pushl  (%eax)
0007  8B4508                    movl   8(%ebp),%eax
000A  FF30                      pushl  (%eax)
000C  684E210000                pushl  $etext+0x14E
0011  E8200B0000                call   etext+0x78
0016  83C40C                    addl   $0xC,%esp
0019  8B4508                    movl   8(%ebp),%eax
001C  8B10                      movl   (%eax),%edx
001E  8955FC                    movl   %edx,-4(%ebp)
0021  8B4508                    movl   8(%ebp),%eax
0024  8B550C                    movl   0xC(%ebp),%edx
0027  8B0A                      movl   (%edx),%ecx
0029  8908                      movl   %ecx,(%eax)
002B  8B450C                    movl   0xC(%ebp),%eax
002E  8B55FC                    movl   -4(%ebp),%edx
0031  8910                      movl   %edx,(%eax)
0033  8B450C                    movl   0xC(%ebp),%eax
0036  FF30                      pushl  (%eax)
0038  8B4508                    movl   8(%ebp),%eax
003B  FF30                      pushl  (%eax)
003D  6866210000                pushl  $etext+0x166
0042  E8EF0A0000                call   etext+0x78
0047  83C40C                    addl   $0xC,%esp
004A  EB00                      jmp    swap+0x4c
004C  C9                        leave
004D  C3                        ret
004E  55                        pushl  %ebp
004F  8BEC                      movl   %esp,%ebp
0051  50                        pushl  %eax
0052  EBAE                      jmp    swap+2

0000  EB51          main:       jmp    main+0x53
0002  E837FFFFFF                call   gotvalue
0007  8945FC                    movl   %eax,-4(%ebp)
000A  E82FFFFFFF                call   gotvalue
                                                    /* DETECTION MECHANISM 3
0007  EB02                      jmp    over3        /*  JUMP OVER
```



```
0011  CC                        int3              /* SEED
0012  CC                        int3              /* SEED
                    over3:
0013  8945F8                    movl  %eax,-8(%ebp)
0016  FF 75 F8                  pushl -8(%ebp)
                                                  /* DETECTION MECHANISM 4
0019  EB01                      jmp   over4       /* JUMP OVER
001B  CC                        int3              /* SEED
                    over4:
001C  FF 75 FC                  pushl -4(%ebp)
001F  68 7D 210000              pushl $otext+0x17D
0024  E8C00A0000                call  otext+0x78
0029  83C40C                    addl  $0xC,%esp
002C  8D45F8                    leal  -8(%ebp),%eax
002F  50                        pushl %eax
0030  8D45FC                    leal  -4(%ebp),%eax
0033  50                        pushl %eax
0034  E8 7A FF FFFF             call  swap
                                                  /* DETECTION MECHANISM 5
0039  EB0B                      jmp   over5       /* JUMP OVER
003B  CC                        int3              /* SEED
003C  CC                        int3              /* SEED
003D  CC                        int3              /* SEED
003E  CC                        int3              /* SEED
003F  CC                        int3              /* SEED
0040  CC                        int3              /* SEED
0041  CC                        int3              /* SEED
0042  CC                        int3              /* SEED
0043  CC                        int3              /* SEED
0044  CC                        int3              /* SEED
0045  CC                        int3              /* SEED
                    over5:
0046  83EC08                    addl  $8,%esp
0049  FF 75 F8                  pushl -8(%ebp)
                                                  /* DETECTION MECHANISM 6
004C  EB01                      jmp   over6       /* JUMP OVER
004E  CC                        int3              /* SEED
                    over6:
004F  FF 75 FC                  pushl -4(%ebp)
0052  E839FFFFFF                call  power
```

```
0051  83C408                    addl  -8,%esp
0054  50                        pushl %eax
0055  689D210000                pushl $otext+0x19D
005A  E8940A0000                call  otext+0x70
005F  83C408                    addl  $8,%esp
0062  C9                        leave
0063  C3                        ret
0064  55                        pushl %ebp
0065  8BEC                      movl  %esp,%ebp
0067  83EC08                    subl  $8,%esp
006A  EB96                      jmp   main+2
```

Other routines are referenced in this code, but for the purpose of analysis the assembler program is considered to be complete.

Appendix E

PUBLICATIONS

APPENDIX E

PUBLICATIONS

E.1. Introduction

To date five papers have been published in connection with the research presented in this thesis. They are as follows,

Paper 1 : Wingate, G.A.S. & Preece, C., *Transient Fault Recovery Assessment in 8 and 16 Bit Microprocessor Based Controllers in Embedded Systems.*, Microprocessing and Microprogramming, Vol. 24, pp 775-782, 1988.

Paper 2 : Wingate, G.A.S. & Preece, C., *Performance Evaluation of a New Design-Tool for Microprocessor Transient Fault Recovery.*, Microprocessing and Microprogramming, Vol. 27, pp 801-808, 1989.

Paper 3 : Wingate, G.A.S. & Preece, C., *Fault Tolerance for Microprocessor-Based Controllers Susceptible to Transient Disturbances.*, IEE Digest 1989/111, pp 3/1-3, 1989.

Paper 4 : Wingate, G.A.S. & Preece, C., *Fault Tolerance for Uniprocessor Systems.*, IEE Digest 1990/176, pp 4/1-5, 1990.

Paper 5 : Wingate, G.A.S. & Preece, C., *Analysis of Failure Data Collected From a TMR Microprocessor Controller.*, Microprocessing and Microprogramming, Vol. 32, pp 861-868, 1991.

The first, second, and last paper listed were presented at the international EUROMICRO '88, EUROMICRO '89, and EUROMICRO '91 conferences held in Zürich (Switzerland), Köln (West Germany), and Vienna (Austria) respectively. The third paper was presented at the IEE Colloquium "Control Systems Software Reliability for

Industrial Applications." organized by the Automation and Control Systems Group C13 in London, October 1989. The fourth paper was presented at the IEE Colloquium "System Architectures for Failure Management." organized by the Control Techniques and Applications Group C9 in London, December 1990.

PUBLISHED PAPERS NOT FILMED FOR COPYRIGHT REASONS

P. 299 to end.

North-Holland
Microprocessing and Microprogramming 24 (1988) 775–782

TRANSIENT FAULT RECOVERY ASSESSMENT IN 8 AND 16 BIT MICROPROCESSOR BASED CONTROLLERS IN EMBEDDED SYSTEMS

G.A.S. Wingate and C. Preece

United Kingdom
School of Engineering and Applied Science
University of Durham

Keywords : Transient Fault Tolerance, Fault Recovery, Microprocessors, Industrial Controllers, Embedded Systems.

Microprocessors used in embedded systems for industrial control applications are often subject to transient disturbances. This can cause system failure unless fault tolerance can be introduced into the design. This paper discusses software design techniques for enhancing fault tolerance in small digital systems. A *metric* is proposed for assessing different designs, and its influence on MTTF is illustrated.

1. INTRODUCTION

Modern industrial control systems are increasingly based on digital circuits incorporating microprocessors. In particular, the use of microprocessor based digital controllers in embedded systems provides an example of a low cost application where simple architectures are preferred. However when replacing analogue circuitry with digital systems, it is important to note that the digital replacements may be more susceptible to catastrophic failure from transient disturbances.

Microprocessor based industrial controllers can enter a state of erroneous execution due to the effects of a transient disturbance. The subsequent execution history depends on the particular microprocessor architecture and the system configuration. In many embedded industrial applications there is a requirement for high reliability, and this can be enhanced by attention to software design. Further improvement can be achieved by integration of hardware and software methods. This problem has traditionally been tackled by electrical shielding techniques. The methods presented here provide added protection by increasing the probability of recovery once a transient fault has occured.

Studies have been published elsewhere, and are referenced below, showing how the probability of recovery can be determined and subsequently enhanced, for systems based on 8-bit microprocessors. Techniques have been proposed for assessing the probability of fault recovery following certain classes of transient disturbance. In this paper this approach is extended to consider 16-bit processors, and in particular the Motorola M68000 series. In order to compare different processors, an overall measurement parameter or *metric* is defined as the probability of executing an instruction which will cause an ordered re-entry to the program. This metric is used to discuss the implications of processor choice for digital controller design.

This paper shows how the inherent properties of 16-bit microprocessors can be utilised to improve the probability of recovery from a transient fault.

The concept of Mean Time To Failure for a system, commonly used for hardware failure estimation, is adapted to provide a measure of transient fault recovery capability.

2. CLASSES OF TRANSIENT DISTURBANCE

The industrial environment is a source of transient disturbances many of which are derived from power supply transients and electromagnetic radiation. It is critical that industrial digital systems in applications such as real-time monitoring and control, have a minimal possibility of complete system failure from transient disturbances. Digital systems are more prone to such a failure than analogue systems which tend to filter the disturbances.

Practical observations and experiments have shown that transient disturbances can cause an erroneous jump in the execution of a program, due to corruption of the program counter. This may be caused by direct corruption of the program counter, of the bus signals, or by data errors which lead to corruption

of stored addresses. The statistical calculation of the probability of recovery following an erroneous jump provides a method of quantifying fault tolerance, and leads to the definition of a recovery metric, which enables recovery strategies to be assessed qualitatively.

Execution at any address results in the processor entering one of a fixed number of classified states. The probability of reaching a particular state may be calculated, based on the proportion of the instruction types in the instruction set, and their distribution in the microprocessor. Some of these states lead to further erroneous execution, whereas others allow an ordered recovery to take place.

Within the system memory, areas can be defined which have different characteristics dependent on the defined utilisation of that memory area. For the purpose of statistical analysis, the memory is divided into distinctive areas, and models are derived for erroneous execution in these areas. The models for 16-bit microprocessors follow the same principles as those developed for 8-bit microprocessors. The differences are due to the inherent architecture and instruction word-length of the 16-bit machines. The common model allows comparisons between machine types to be made. Once figures for recovery probabilities have been calculated for a particular design, techniques for improving the metric can be proposed. The aim of the design technique is to maximise the probability of an ordered recovery after an erroneous jump.

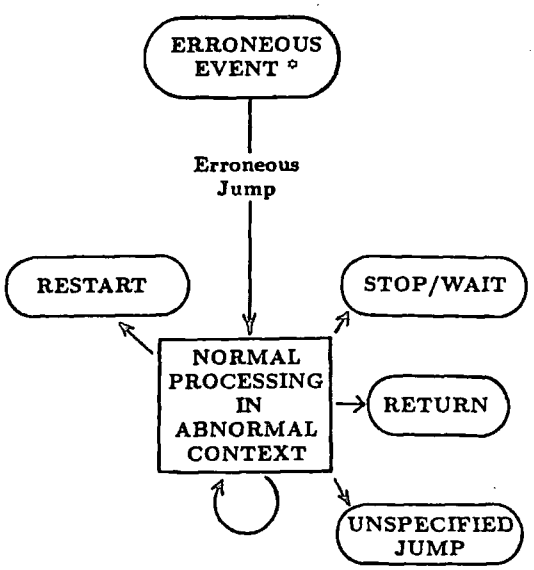


Figure 1. 8-Bit Microprocessor Erroneous Execution Model.

(* manifested transient fault)

3. REVIEW OF 8-BIT CONCEPTS

The concepts used in the study of 8-bit machines are reviewed here, in order to illustrate the comparison between 8- and 16- bit processors. A full description will be found in Reference 1.

Take the M6800 microprocessor which has an addressing range of 64K bytes using a 16-bit address bus. An erroneous jump from a running program to a random address in the address space results in an entry to one of five states as illustrated in Figure 1, the state reached being determined by the value of the data at the particular address. The probability of reaching one of the five states can be calculated knowing the proportion of particular instructions within the instruction set.

Of the 256 possible op-codes in the M6800 microprocessor, not all are defined. Those that are undefined have various state outcomes. The probabilities of entering each state after an erroneous jump are shown in Figure 2 for the M6800 when the contents of the memory area are assumed to be random. The probabilities of entering each state change as further instructions are executed following the original jump as shown in the figure, the plotted lines represent the boundaries between states.

The initial probability of entering a particular state, P_S , is given by

$$P_S = \frac{N_S}{N_T} \tag{1}$$

where N_S is the number of op-codes corresponding to this state, and N_T is the total number of possible op-codes.

The probability that K instructions will be executed before a state is reached where a jump out of ordered processing will occur is given by $P_J(K)$ where

$$P_J(K) = (1 - P_J)^{(K-1)}.P_J \tag{2}$$

from Reference 2.

An ordered recovery following a transient event implies the execution of a restart instruction. The probability of executing such an instruction, illustrated in Figure 2, is low because of the small proportion of restart op-codes in the M6800 instruction set. Similar results *loc cit* for other 8-bit processors serve to confirm this conclusion for this class of processor.

Techniques for improving the probability of recovery in 8-bit systems have been reported in Reference 3. It is shown below that these methods are much more effective when applied to 16-bit processors.

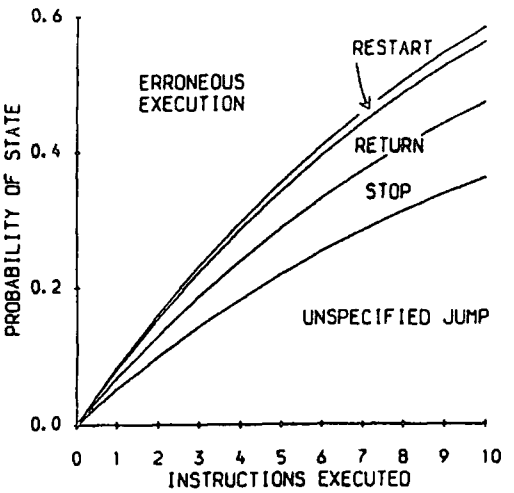


Figure 2. 16-Bit Model Erroneous Execution State Outcome.

4. COMPARISON WITH THE M68000 SERIES

4.1 State and Type definitions

The microprocessors considered here in the Motorola M68000 family are the M68000, M68010, and the M68020. These microprocessors have addressing capabilities of between 24 and 32 bits, and have 16 bit data buses. The instruction set is based upon a 16-bit instruction word. This gives the microprocessors a possible instruction set of 65536 instructions.

The M68000 family of microprocessors are micro-coded. That is, the 16-bit op-code is presented to an *execution unit*. This unit is effectively a ROM to which the 16-bit op-code is an address. The ROM then releases, for every possible address variant, an appropriate sequence of micro-codes which will carry out the requested operation. Illegal and undefined instructions are treated in exactly the same way as legal and defined instructions. Illegal and undefined instructions have a specified operation: an exception call.

This fact is particularly significant in terms of fault recovery as execution of any invalid op-code leads to an exception, which in turn, can lead to an ordered recovery through an exception service routine.

The numbers of defined and undefined instructions for the M68000 series are shown in Table 1, where they are compared with equivalent figures for the M6800.

Table 1. Number of defined and undefined instructions for the MC6800 and M68000 microprocessor family.

Microprocessor	No. of instructions	
	defined	undefined
MC6800	197	59
M68000	43342	22194
M68010	43521	22015
M68020	46595	18941

The instruction types can be classified as shown in Table 2.

Table 2. Instruction Type Classification

Non - Jump
Restart (software interrupt, software exception)
Return
Stop (wait)
Undefined Instruction
Unspecified Jump

We consider an "op-code state" to be the state resulting from the interpretation of one of the instruction types as follows:

Non-Jump - leads to the *program counter* pointing to the location following a valid single or multi-byte instruction.

Restart - leads to a jump to a predefined location in the memory map (by exception).

Return - leads to a jump to an address held in a stack.

Stop/Wait - leads to cessation of processing; and requires an interrupt or hardware reset to exit from this state.

Undefined Instruction - leads to a *restart* in the M68000 series because of the undefined instruction exception feature.

Unspecified Jump - leads to a jump to a new location determined by local memory contents.

Of these states, only two are capable of providing an ordered restart, the *restart* and the *undefined instruction* exception. A diagram showing the possible states

following an *erroneous jump* in a 16-bit microprocessor is shown in Figure 3. This diagram can be compared to the corresponding diagram for 8-bit microprocessors shown in Figure 1.

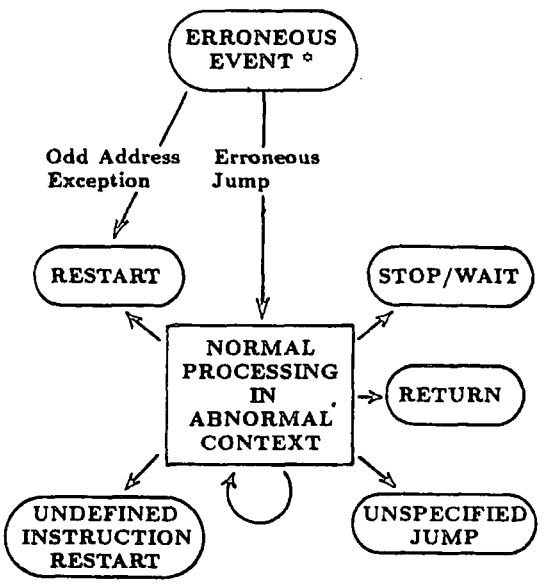


Figure 3. 16-Bit Microprocessor (M68000 Family) Erroneous Execution Model.
(* manifested transient fault)

4.2 Microprocessor Model

In developing the model of microprocessor operation subsequent to a transient event leading to an *erroneous jump*, the assumption is made that there is an equal probability of the *program counter* containing any address within the memory map.

The memory map is divided into notional areas. Assumptions will be made later about the properties of particular areas which will modify the statistical evaluation of state outcome. The categories are shown in Table 3.

Table 3. Memory Map Categories

Memory Map Category
Input/Output Reserved Area.
Program Area.
Data Area.
Unused Area.

If an initial assumption is made that the data area of the memory contains random data, then comparison can be made between the 8-bit and 16-bit machines, by calculating the probability of entering a particular state as before. However, an important feature of the M68000 leads to an addition to the equation. Any attempt to fetch an instruction from an odd address leads to an exception. Assuming that the value of the the corrupted *program counter* is random, then the probability that the *program counter* holds an odd address is 0.5.

All probability calculations for the M68000 series therefore apply to even addresses, recovery is guaranteed for all odd address references. A diagram of state probability for the M68000 following an erroneous jump to random data area is shown in Figure 4. Comparison of this diagram with Figure 2 shows the enhanced probability of restart in the M68000. The percentages of different instruction types in the microprocessor instruction sets are given in Table 4.

Table 4. Instruction Type Percentages in Microprocessor Instruction Sets.

Instruction Type	M68000	M68010	M68020	M6800
Non-Jump	57.825	58.095	59.327	91.600
Restart	1.970	1.970	3.683	0.300
Return	0.003	0.006	0.006	1.300
Stop	0.002	0.002	0.002	1.600
Undefined Instruction	33.865	33.592	28.902	*
Unspecified Jump	6.335	6.335	8.080	5.200

* The M6800 has 59 undefined instructions, but unlike the MC68000 family of microprocessors, these instructions do not lead to exception and recovery. They have therefore been grouped with the other instruction types dependant on their effective action.

5. EXECUTION OF A PROGRAM IN AN EMBEDDED SYSTEM

For many industrial control and monitoring applications, the program may occupy only a small proportion of the addressable memory map. When a transient event occurs, the corrupted program counter may point to any location, whether used or not, in the addressing range. We will refer to the area occupied by the program, data storage, and memory mapped I/O as the *used area* , and the remainder of the map as the *unused area*. If the unused area is filled with data which are interpreted as restart instructions, or other instructions which would generate exceptions,

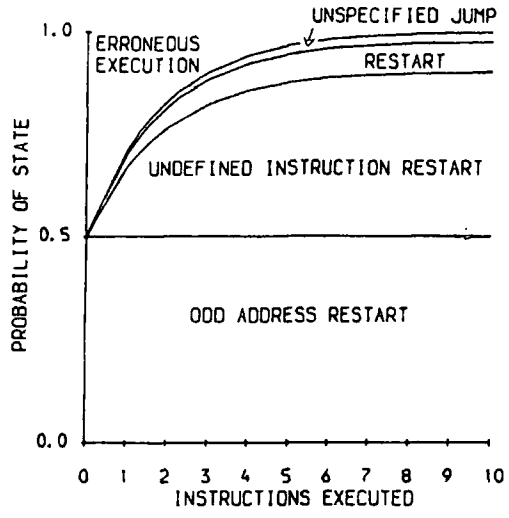


Figure 4. 16-Bit Model Erroneous Execution State Outcome.

then the probability of recovery is further improved. Methods of achieving this have been discussed in Reference 3, the most powerful of which is bus-biassing. This consists of external circuitry which asserts a bit pattern on the data bus when any unused memory address is accessed during an instruction fetch. If the bit pattern is chosen to force an exception, then all references to unused areas provide recovery.

In this situation the probability of recovery depends, not only on the proportions of restart instructions in the instruction set, but also on the ratio of used to unused memory in the whole addressable memory area.

The concept of Mean Time To Failure, MTTF, used in hardware reliability calculations, can be adapted for this work. It provides a method of comparing the improvement in reliability brought about by fault recovery with other hardware and software methods in embedded systems. It also enables comparisons between designs using different microprocessors to be made.

6. MTTF FOR A SYSTEM SUBJECT TO TRANSIENT EVENTS

Let the sample space E, comprise a set of events corresponding to erroneous jumps in the running program. Let $E_r \in E$ where E_r is an event leading to a recovery, and $E_f \in E$ where E_f is an event leading to a failure. We can also state that $E_r \cup E_f = E$ and $E_r \cap E_f = \emptyset$.

Let the probability of event E_r be $P(E_r)$ and the probability of event E_f be $P(E_f)$, which leads to

$$P(E_r) + P(E_f) = 1 \tag{3}$$

If we assume that transient events occur at a rate of k events per hour, then the rate of failures per hour is given by Λ where

$$\Lambda = k.P(E_f) \tag{4}$$

Assuming an exponential probability distribution function $f(t)$, then

$$f(t) = e^{-\Lambda t} \tag{5}$$

$$f(t) = e^{-k.(P(E_f)).t} \tag{6}$$

and

$$MTTF = \frac{1}{k.P(E_f)} = \frac{1}{k.(1 - P(E_r))} \tag{7}$$

Equation 7 gives a value of MTTF for the system when subjected to transient events at the rate of k per hour, and where the probability of recovery from any single event is $P(E_r)$. It allows a comparative assessment to be made for software recovery techniques in a form which can be related to MTTF calculations for permanent faults in digital systems.

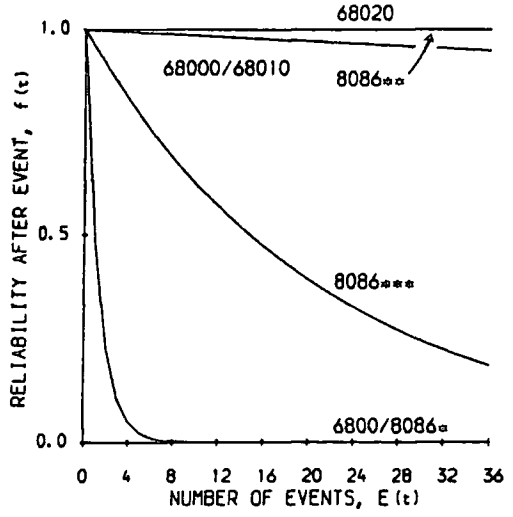


Figure 5. Microprocessor Reliability.

(for key see Table 5)

Let us take an example of a system having a *used area* of 48K bytes. If we assume, as a worst case, that any erroneous jump into the 48K byte area will result in system failure, but any jump into the *unused area* is recoverable, then the effect of different processors on the reliability can be seen in Figure 5. These curves are drawn with a normalised time base.

It is helpful to consider a numerical example to illustrate the point. If we assume that the system is subjected to transient events at a rate of, say, $1.4\,10^{-3}$ per hour, or approximately once per month. Table 5 shows the effect on MTTF of variation in $P(E_r)$. The MTTF for this event rate with no recovery, (i.e. $P(E_r) = 0$), is 714 hours.

Table 5. MTTF for different microprocessors.

Processor	$P(E_r)$	MTTF
M6800	0.250000	952 hrs = 39 days
8086 *	0.250000	952 hrs = 39 days
8086 **	0.999969	23405714 hrs = 2672 yrs
8086 ***	0.953125	15238 hrs = 21 mths
M68000	0.998535	487567 hrs = 56 yrs
M68010	0.998535	487567 hrs = 56 yrs
M68020	0.999994	119047619 hrs = 13590 yrs

* assuming corruption of the *program counter* only, and that the program is contiguous.

** assuming corruption of the *segment register* only, and that the program is contiguous.

*** assuming corruption of the *program counter* and *segment register* are taken together as a single register.

In the case of the 8086, two registers are involved in specifying the address of any instruction. Corruption of each register has been treated separately. A third case has been considered (assuming that the 8086 could be considered to have a single rather than a multiple, *program counter*) so that a comparison for a microprocessor with the same addressing range as the 8086 can be made. Table 5 illustrates the probability of recovery and the MTTF for a range of microprocessors.

It can be seen that once the value of $P(E_r)$ approaches a value of 0.99 or better, a small increase in value can bring a large improvement in MTTF.

As the proportion of *used area* increases, the change in MTTF can be calculated as, in general,

$$MTTF = \frac{(\text{ total area in bytes })}{k.(\text{ used area in bytes })} \qquad (8)$$

This calculation assumes that any entry into the *used area* leads to failure. However this is not actually the case. As mentioned above, different areas of the memory map have different properties, and more accurate figures for recovery probability can be determined for these areas.

7. MODIFICATION TO MTTF ESTIMATE BY MEMORY CATEGORIES

Within the *used area* of the memory map, that is the area containing the program with its data area, and mapped I/O addresses, distinctly different properties of the contents of these areas can be defined.

The program area contains program code consisting of op-codes and operands. As a first approximation we may assume that there are no restart codes in the program area. This means that any erroneous jump to a program area results in $P(E_r) = 0$.

The data area on the other hand, consists of numerical data which bear no relation to an ordered sequence of instructions. It is therefore possible to assume a random distribution of data values. We here include I/O mapped registers in the data area. For the M68000 this gives a value of $P(E_r) = .35835$ within the data area for all even addresses.

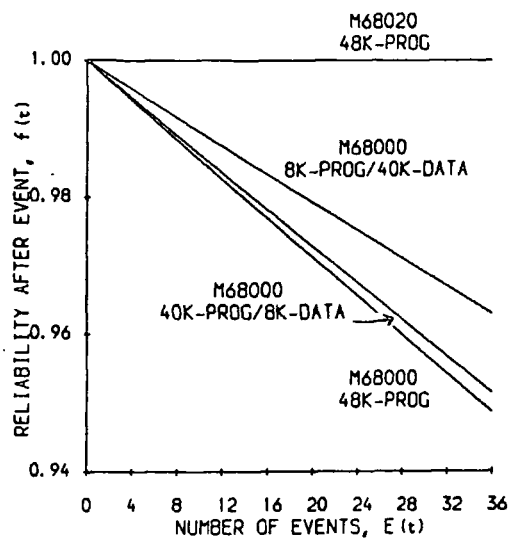


Figure 6. Improved M68000 Microprocessor Reliability.

The effect of this different treatment for different used areas of memory is shown in the examples given in Figure 6. A typical program area of 48K is assumed in one case to comprise 40K of program and 8K of data, in the second case the proportions are reversed. These are compared with the case of a 48K program with no data area. If we assume an event rate of approximately 1 per month as in the examples above, then the MTTF for these applications can be evaluated. The results for implementation on the M68000 are shown in Table 6.

Table 6. MTTF for various Program/Data ratios in M68000

Program/Data	$P(E_r)$	MTTF
48K / 0K	0.998535	487567 hrs = 56 yrs
40K / 8K	0.998617	516476 hrs = 59 yrs
8K / 40K	0.998948	678979 hrs = 78 yrs

Any pattern of memory use can be analysed to give a more accurate value of MTTF for a particular application program.

8. ENHANCEMENT OF RECOVERY PROBABILITY BY DESIGN

A number of design options present themselves as candidates for improving the recovery metric $P(E_r)$. It is clear that the aim is to increase the number of codes in the memory which produce vectored restarts. The treatment of *unused* memory has already been referred to. All unused locations should be filled with restart or exception codes, either by bus-biassing, or by special EPROMs. Partial decoding of the EPROMs can reduce the number required in a particular system. This technique is universally applicable, and does not depend on the detail of the program or the application; it may be particularly advantageous where the microprocessor has a multiplexed bus.

The detailed treatment of data and program areas of the memory is dependent on the properties of the code for a particular application. Some general rules can, however, be formulated.

As a first approximation, if data areas can be considered to contain random numbers, then the proportions of codes given by Table 4 apply. The inherent metric is a function of the restart and undefined instructions. However, over 50% of the instruction types are in the non-jump category, that is, execution proceeds beyond them in sequence. This observation introduces the possibility of utilising this property to force further restarts by seeding the data areas with recovery traps, sequences of codes spread throughout the data area. As well as providing direct recovery if an erroneous jump lands on a trap, it also enhances the probability of recovery after execution of a non-jump instruction.

The treatment of program areas is quite different. Here the codes are valid instructions, and entry to any program area has a high probability of resuming valid code execution. (Reference 1). Seeding the program area is application dependent, and the programmer needs to have this in mind when writing the code. Alternatively this function might be implemented by a high level compiler. Two examples of programming techniques involving operands in the M68000 series will be sufficient to illustrate the point.

i) A proportion of a program area may contain codes representing addresses of operands. These addresses point to items of data storage. If the data is stored at addresses which themselves represent invalid instruction codes, then erroneous execution of any of these operands in the program area will force restarts.

ii) An alternative form of addressing can also produce the same result. If data is referenced using backward relative addressing, then execution of the operands containing the negative data offset will cause invalid op-code exceptions in the M68000 and M68010. This is because setting the four most significant bits in the word produces a code which is interpreted as an exception in these processors.

9. DISCUSSION

Transient faults can cause a microprocessor system to experience corruption of the program counter causing erroneous jumps to random locations in memory. This brings a loss of control unless some mechanism for recovery is present. The probability of recovery can be enhanced if a restart can be initiated after the fault. The inherently larger addressing space of 16-bit microprocessors provides a major improvement in the recovery metric, as long as all the unused memory is designed to contain restart instructions, or to initiate exceptions, (software interrupts).

The concept of Mean Time To Failure can usefully be applied to systems subjected to transient disturbances. The definition of MTTF incorporates the probability of achieving a restart state at any event $P(E_r)$, and also the rate of events k per hour.

The method enables the reliability of microprocessor based embedded systems to be assessed, and different designs to be compared. It also provides a statistical basis for developing the techniques for enhancing the fault tolerance of small systems which are described in Reference 3.

The results suggest that software recovery for transient events can provide an additional means of protection for microprocessor based systems in addition to that traditionally provided by external hardware watchdog circuits. In selected 16-bit machines the figures for MTTF can be high even for onerous transient event rates.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the support of the UK Science and Engineering Research Council and the British Gas Engineering Research Station, Killingworth, Newcastle upon Tyne. They would also like to thank Dr. R.G. Halse, of Westinghouse Signals Ltd., Chippenham, Wilts. for his suggestions and helpful comments on the paper.

REFERENCES

[1] Halse R.G. *Fault tolerance in digital controllers using software techniques.* Ph.D. Thesis. University of Durham, England. 1984

[2] Halse R.G. and Preece C. *Erroneous execution and recovery in microprocessor systems* Software and Microsystems 4 No. 3. June 1985. pp 63-70.

[3] Halse R.G. and Preece C. *Recovery assessment after microprocessor transient disturbances* in *System Fault Diagnostics and Related Knowledge-Based Approaches* Vol 2. pp 383-397. S.Tzafestas et al.(eds) 1987 D. Reidel Publishing Co.

North-Holland
Microprocessing and Microprogramming 27 (1989) 801–808

PERFORMANCE EVALUATION OF A NEW DESIGN-TOOL FOR MICROPROCESSOR TRANSIENT FAULT RECOVERY

G.A.S. Wingate (UK) & C. Preece (UK)

School of Engineering and Applied Science,
University of Durham, DH1 3LE. England.

Keywords : Transient Fault Tolerance, Fault Recovery, Design Tool, Embedded Systems, Microprocessor Controllers.

Approach : Evaluation.

A model of microprocessor erroneous behaviour has led to the development of a new design tool to automate the introduction of transient fault tolerance into program code. The design tool, PARUT (Post-programming Automated Recovery UTility) provides a method of enhancing existing program code to optimise the recovery capability following a transient disturbance. The tool can be used to implement a number of different recovery strategies, some of which may involve additional hardware. The paper examines the performance of the design tool for a range of techniques.

1. INTRODUCTION

Modern designs of industrial control systems incorporate microprocessors for control and monitoring purposes. The versatility that microprocessor based controllers offer to the designer, and the flexibility that customised software provides, makes them attractive in many industrial situations.

However, industrial environments are often harsh, falling short of the ideal for computer systems. In particular, microprocessor operation can be corrupted by externally generated transients events such as electrical power transients [1] and electro-magnetic radiation [2, 3, 4]. Even in 'benign' operating conditions transients have been observed to cause between 80% and 90% of digital system failures [5, 6, 7, 8]

In analogue systems these transients go through a 'filtering' process which generally means that the control function is not lost. Digital systems are much more liable to lose all control function following a transient disturbance. It is important that digital design should incorporate mechanisms for recovery in the event of erroneous execution.

Transient events considered in this paper are those which lead to corruption of data on the bus, or in the memory, or registers of a microcomputer system. While such corruption can lead to erroneous behaviour of the system, no permanent hardware damage is incurred, and if control of the computational process can be re-established, then this permits the possibility of overall recovery of the control system.

A number of well known techniques are available to reduce the probability of failure from transients. These include both hardware and software enhancement and usually incorporate some form of redundancy. A technique such as hardware modular redundancy with voting will prevent many transient failures but involves considerable hardware overhead [9]. Watchdogs circuits [10], although commonly used, themselves suffer from transients. They also introduce a performance overhead by constantly interrupting microprocessor operation.

This paper includes a comparison of two other recovery techniques which aim to improve transient fault tolerance by the inclusion of both hardware and software enhancements. These have been applied to 8-bit, 16-bit, and 32-bit microprocessors [11, 12, 13]. The hardware modification is simple with a very low overhead. The software enhancement has involved up to 60% increase in memory requirement, but this figure is very much code dependant. A post-programming utility has been built which automates the method of enhancing software to improve transient fault tolerance. The utility is described, and its performance with transient fault tolerant techniques is evaluated.

2. ERRONEOUS BEHAVIOUR

We consider erroneous behaviour to be initiated by corruption of the microprocessor's internal state which leads to the corruption of the program counter. It has been suggested that 25% of transients affecting digital systems will corrupt the internal state [6]. The erro-

neous behaviour produced by a microprocessor is characterised as a sequence of erroneous execution states terminated either by recovery or catastrophic failure.

Execution states can be defined in terms of the outcome of each operation. The possible states are defined as follows :

Non-Jump - leads to the program counter pointing to the location following a valid instruction.

Restart - leads to a jump to a predefined location in the address space.

Unspecified Jump - leads to a jump to a new location in the address space determined by local memory contents.

Return - leads to a jump to an address held in a stack.

Stop/Wait - leads to a cessation of processing ; and requires an interrupt or hardware reset to exit from this state.

Not all possible instruction bit patterns in a microprocessor are necessarily defined. In such cases these undefined instructions can, when executed, result in any of the defined states. It should be noted however that because no specification is available for these undefined instructions, manufacturers are not obliged to ensure every die batch produces the same operation for each of the undefined instructions. However, the problem does not arise in all microprocessors. In the case of the the Motorola 68000 family, for example, the execution of all undefined instructions results in an 'exception' (or software interrupt) leading to the 'restart' state as defined above.

For purposes of discussion, we assume that a transient fault will cause random corruption of the program counter contents. The instruction following the transient event will be fetched from a location pointed to by the corrupted contents of the program counter resulting in a jump in the control flow of the executing software. This random jump is termed the Initial Erroneous Jump (IEJ). We define execution following the IEJ as 'erroneous execution'. The content of the memory at the new execution location is fetched and executed as if it were a valid instruction. The outcome of this erroneous execution will result in a new behavioural state. Execution of a non-jump type instruction continues linear erroneous execution. If the new state is a jump type (but not a 'restart'), then a Subsequent Erroneous Jump (SEJ) will occur. In order to achieve controlled recovery it is necessary to maximise the probability of a 'restart' type instruction being executed as soon as possible after the inception of erroneous execution.

3. MODEL

A model of erroneous execution has been presented elsewhere [11, 12, 13]. and is reviewed briefly here.

The probability of a jump type outcome after k instructions have been executed in a random area [11] is given by

$$P_J(k) = P_{NJ}(k-1).P_J \qquad (1)$$

where P_J and P_{NJ} are the initial probabilities of a jump and non-jump outcome. If the simplifying assumption is made that the content of the memory at the jump target is random, then the probabilities for individual outcomes can be calculated from the distribution of their associated instructions within the instruction set, and multiplying by $P_J(k)$.

Improvement in the probability of recovery following an initial erroneous jump (IEJ) has already been discussed in a previous paper [12] where unused address space is filled with restart type instructions. Major improvements were shown for a range of 8, 16, and 32 bit microprocessors. For erroneous execution in the 'used areas' of memory, the probability of a number of random subsequent erroneous jumps (SEJ) landing within the used area is given by

$$P_{SEJ}(\text{Used Area}) = \left\{ \frac{\Sigma^J \Sigma^L \left(\frac{\text{No. addressed bytes in used area}}{\text{No. bytes in address range}} \right)}{N_J.N_L} \right\} \qquad (2)$$

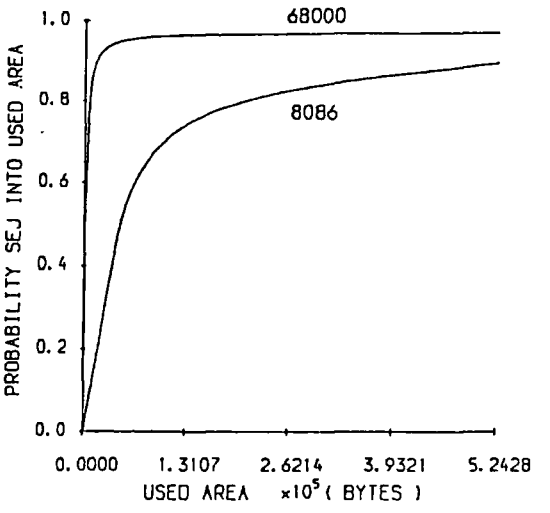


Figure 1 : SEJ Characteristic.

where the summations to J, and L, map each jump type instruction and, every location in the used area respectively. N_J is the number of jump type instructions in the instruction set. N_L is the number of locations in the used area.

The SEJ characteristic for the Motorola 68000 and Intel 8086 is shown in Figure 1. This graph emphasises the importance of implementing a recovery technique within the used area. If no recovery technique is implemented then there is a high probability of an extended period of erroneous behaviour consisting of many SEJs. A detailed analysis of this characteristic is given in Reference 13.

4. PARUT (Post-programming Automated Recovery Utility)

PARUT has been designed to accomplish the following:

i) To analyse the original code and report the IEJ recovery capability inherent in the original code, and the SEJ recovery capability.

ii) To enhance the fault tolerance of the original code by a selection of methods. This usually involves inserting some redundancy into the code. When this is completed, the utility re-aligns the original software control flow which will have been offset at the machine code level by the introduction of redundancy.

iii) To analyse the enhanced code and report the improvement in the IEJ recovery capability, the improvement in the SEJ recovery capability, and the coded area extension overhead required.

The PARUT program has two input requirements, a description of the microprocessor on which the code is to reside, and a copy of the code. Processing this information results in two output streams, a report on the original and the enhanced codes, and a copy of the enhanced codes. An overview of the PARUT program is shown in Figure 2.

In the present version of the PARUT program, detection mechanisms cannot be inserted for SEJs that originate within, and whose destinations lie within, the same instruction [13]. The utility does, however, report these occurrences.

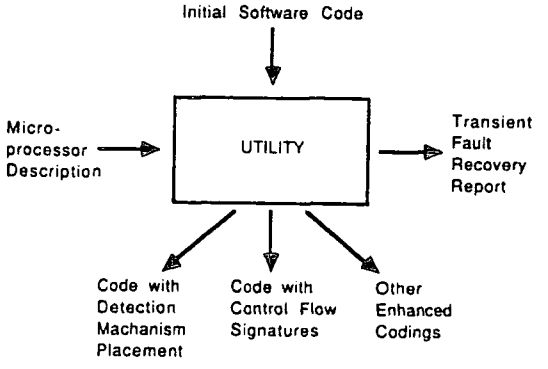


Figure 2 : PARUT Overview.

5. APPLICATION : Motorola 68000

Studies of a range of microprocessor types have suggested that while many features of erroneous behaviour are common to all, [11, 12], detailed analysis requires that each type is considered separately. We consider here the Motorola 68000 microprocessor as a typical example of commonly used 16/32 bit microprocessors. The analysis produced for this microprocessor is specific, but the general approach and results are valid for a range of microprocessors and architectures.

The model for the Motorola 68000 follows from the description of microprocessor behaviour. A particular feature of the Motorola 68000 family of microprocessors is the 'odd address exeption'. The handling routine for this exception can be written so that it directs execution to the recovery routine. With unused even addresses also giving a restart (via a hardware address access guardian), the microprocessor gives an enhanced probability of recovery following an IEJ.

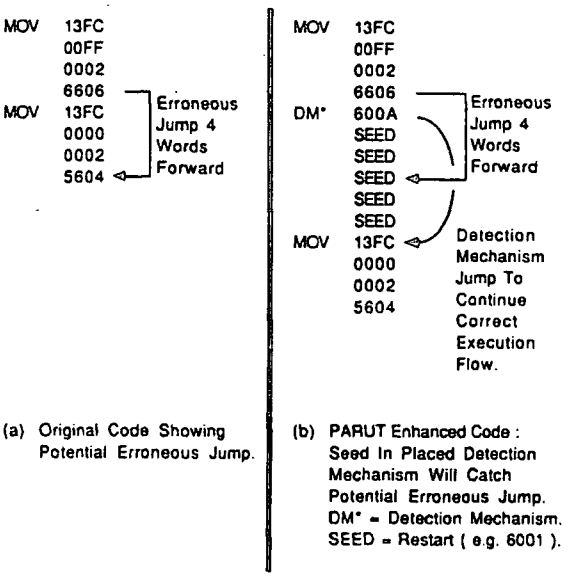


Figure 3 : Example of Detectection Mechanism Placement.

To improve the recovery capability of the microprocessor still further, PARUT can implement detection mechanism placement. The detection mechanism used for the Motorola 68000 is shown in Figure 3. It consists of an initial one-word relative branch instruction over the remainder of the mechanism, so that logical control flow of the correctly executing program is not interrupted. This does however incur an additional processing overhead. The remainder of the detection mechanism consists of five one-word seed instructions. A seed instruction is a software exception instruction which directs execution flow to the recovery routine. Five seed instructions are necessary because the maximum length of an instruction in the 68000 microprocessor is five words. Detection mechanisms must be placed so as not to disrupt correct execution flow. The rule for placement is that where possible each SEJ destination becomes a seed in the detection mechanism, thus ensuring a restart during the next execution cycle, and an increased probability of recovery following an SEJ. The model applicable to the M68000 is shown in Figure 4.

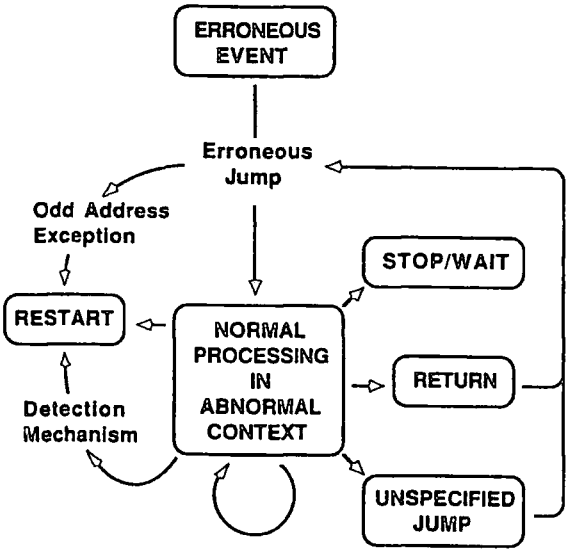


Figure 4 : Erroneous Execution Model.

To illustrate the versatility of the PARUT method a second technique has been implemented, based on proposals in a paper by Schutte and Shen [14] of control flow monitoring using 'Signatured Instruction Streams '. For the purposes of comparison, PARUT has been modified to produce code with embedded signatures. The simulation inserts a single random word immedately following all valid control flow instructions in the original code.

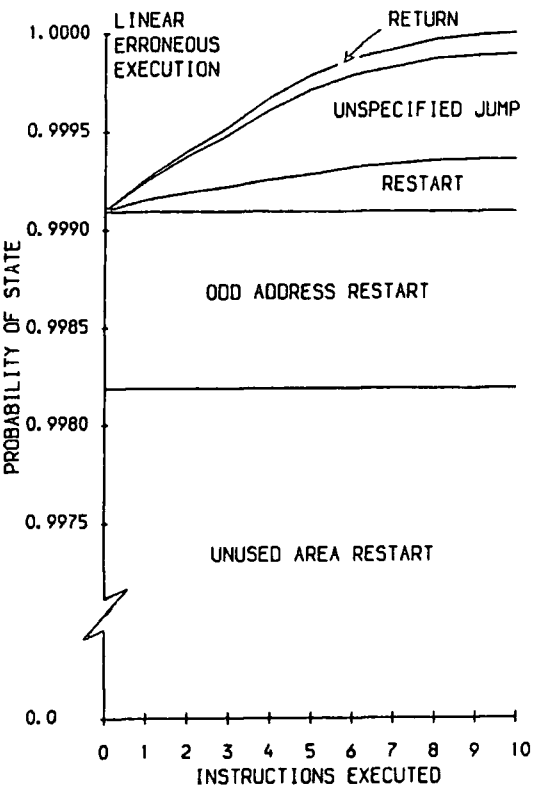


Figure 5 : Execution History Following IEJ (Original Code)†.

6. PERFORMANCE EVALUATION

The results presented here show the effect of the PARUT utility on a particular piece of code. The results are entirely code dependant, and will vary greatly from one program to another. An assessment is made of the improvement in recovery capability achieved by detection mechanism placement.

The performance evaluation of original and PARUT enhanced codings requires examination of the execution histories belonging to each of the two phases of erroneous behaviour. The first phase is defined as that of erroneous execution following an IEJ. The second phase consists of the erroneous execution following each of a number of SEJs until either recovery or catastrophic failure occurs. The example coding is taken from a short Motorola 68000 control program written in assembler code.

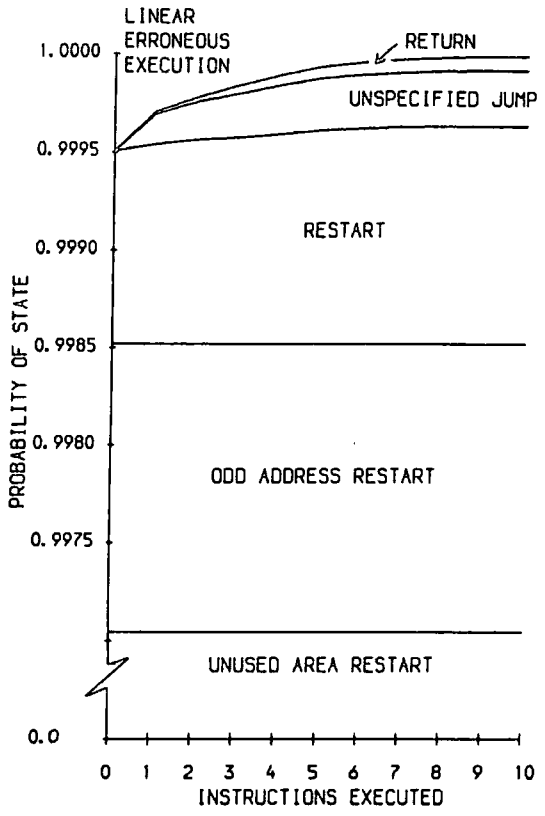


Figure 6 : Execution History Following IEJ
(Code with Detection Mechanisms)†.

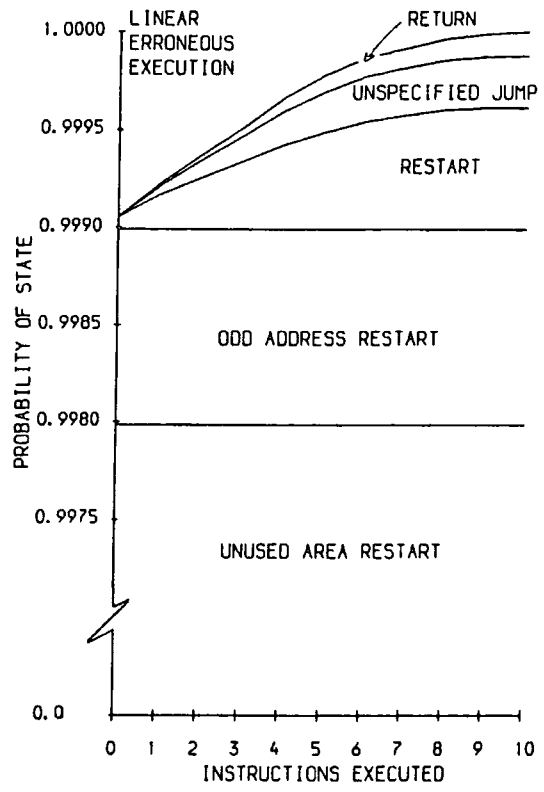


Figure 7 : Execution History Following IEJ
(Signatured Code)†.

6.1 Phase 1 Observations

The execution histories for the original code, the code with detection mechanism placement, and the embedded signature code, are shown in Figures 5, 6, and 7 respectively. These results are for execution histories following an Initial Erroneous Jump.

Both enhanced codings exhibit a lower initial probability of restart. This overhead is due to the extended memory required for the inserted detection mechanisms and embedded signatures.

The code with detection mechanism placement has a smaller probability that the outcome state will be a jump (without restart) compared to the original code. This is due to the probability of an initial SEJ destination being a detection mechanism seed delivering a restart outcome.

The signatured code also shows a decrease in the probability jump (without restart) outcome state. This is because the signature process delivers a restart outcome for any initial SEJ which does not synchronised with valid program flow.

The behaviour of this first phase of erroneous execution, if restart is not achieved, is characterised by a short period of linear execution followed by a further erroneous jump. Figures 6 suggests that the enhanced code with embedded signatures will have the longest period of linear execution, followed by the code with detection mechanism placement, and then the original code. If the first phase of erroneous behaviour is longer than three instructions then the performance of the enhanced code need not necessarily be reduced. However, it is only when the effects of SEJ are taken into account that the overall improvement is clearly apparent.

6.2 Phase 2 Observations

The execution histories for the original code, the code with detection mechanism placement, and the embedded signature code, are shown in Figures 8, 9, and 10 respectively. These results are for execution histories following a Subsequent Erroneous Jump.

Some SEJ destinations cannot be determined due to their use of data which is only specified at run-time. In such instances, for analysis purposes, the destinations

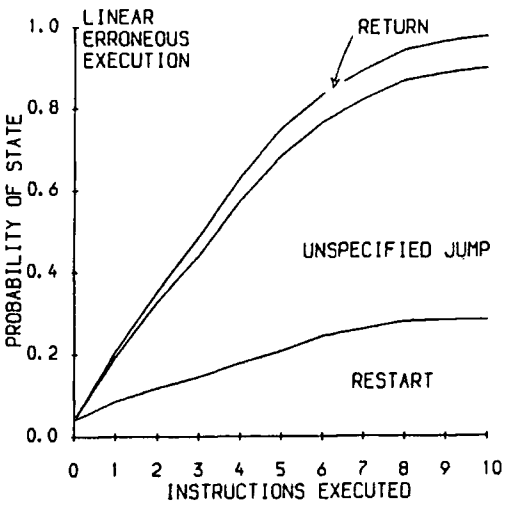


Figure 8 : Execution History Following SEJ (Original Code)†.

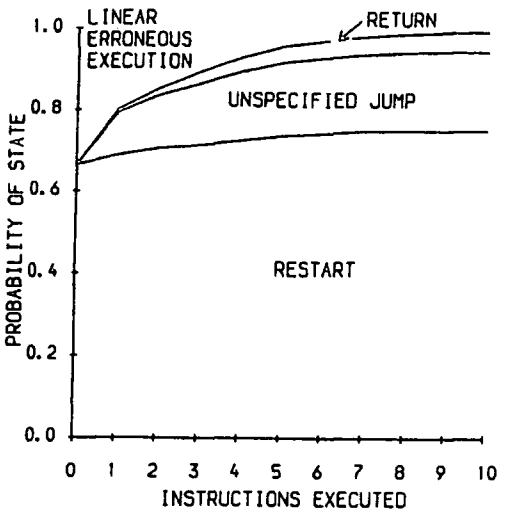


Figure 9 : Execution History Following SEJ (Code with Detection Mechanisms)†.

are assumed to be random. This is important in estimating the proportion of SEJ destinations which lie outside the used area, or at an odd address, and which affect the probability of restart.

The effect of placing detection mechanisms is clearly demonstrated by the increased probability of restart as shown in Figure 9 compared to the original code shown in Figure 8.

In the case of the signature method, an SEJ by an attempted execution of an operand will not complete its operation because of the hardware detection circuitry. On the other hand an SEJ by an attempted execution of an op-code results in re-synchronisation with the program flow.

6.3 Recovery Performance

Recovery is assumed to be achieved through a restart type instruction directing the program flow to a recovery routine.

The absolute probability of restart can be calculated from the information held in both the execution histories for erroneous exceution following an IEJ and SEJ. Let $P_{RT}(k)$ be the absolute probability of restart after k instructions processed following an IEJ. Let $I_{RT}(k)$ and $I_J(k)$ be the probabilities of restart and, jump without restart, after k instructions processed following an IEJ. Let $S_{RT}(k)$ and $S_J(k)$ be the probabilities of restart and, jump without restart, after k instructions processed follwing an SEJ. Finally, let n, x, y, and z be instruction indicies.

$$P_{RT}(w) =$$

$$I_{RT}(w) + \sum^{x}\sum^{n}\sum^{z}\left\{I_J(x).\prod^{n} S_J(y_n).S_{RT}(z)\right\} \quad (3)$$

where $x + \sum^n y_n + z = w$, and $n, x, y, z \geq 0$.

The absolute probability of restart for the different enhanced versions of the program are shown in Figure 11. The performance of the PARUT generated code shows improvement over the original code after one or more instructions have been erroneous executed. The period of erroneous behaviour is reduced leading to a smaller probability of data corruption. This in turn will improve availability, and reduce the chance of catastrophic failure.

Absolute probabilities of restart tends to a value less than 100% because of the probability of resynchronisation with the program flow. If resynchronisation occurs, complementary recovery techniques are required [15, 16].

6.4 Overheads

Detection mechanism placement incurs a memory overhead. This overhead is clearly demonstrated in the reduced initial probability of recovery through restart for erroneous behaviour following an IEJ, see Figure 6. Examination of Figure 11 reveals how this initially reduces the absolute probability of restart of the PARUT enhanced code compared to the original code. However

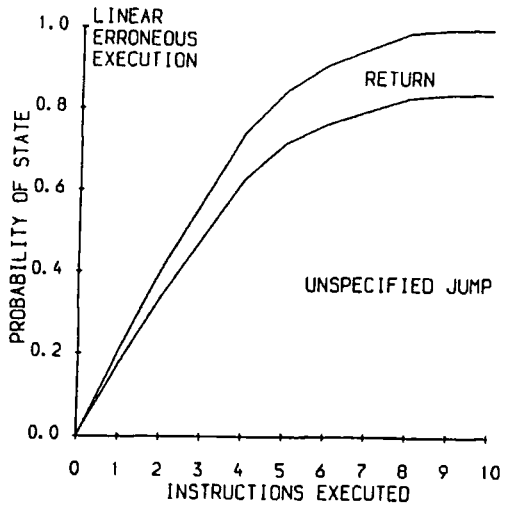


Figure 10 : Execution History Following SEJ (Signatured Code)†.

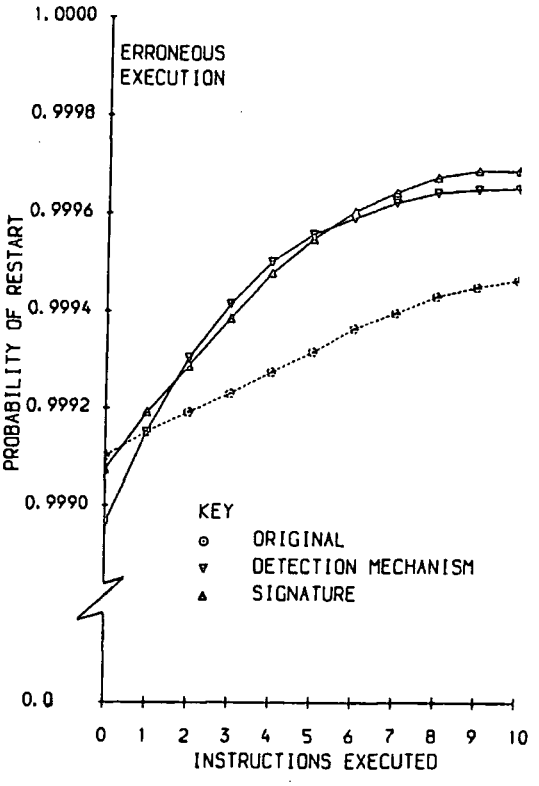


Figure 11 : Absolute Probability of Restart During Erroneous Execution.

we note in Figure 11 that the performance of the enhanced code improves due to the increased probability of restart following a SEJ produced by detection mechanism placement, see Figure 9.

The execution of the detection mechanism jumps over the seeds, during correct program flow, will incur a small processing overhead. The influence of this on the program performance is code dependant.

7. DISCUSSION

The method described in this paper for improving the transient fault recovery capability is valuable for applications which require a high degree of dependability or which are safety critical.

The microprocessor model described above can be extended to other types of microprocessors [11, 12]. The design tool PARUT is based on this model and is therefore widely applicable.

The recovery technique used for erroneous execution following an IEJ requires the unused area to have a 100% detection capability during the processing of the initial instruction. In the case of the Motorola 68000 microprocessor, this can be achieved by additional circuitry that detects whether invalid address lines have been activated, and if so, impresses a bus error exception signal to the microprocessor. This is acceptable if the used area forms a contiguous block in the memory map. If it does not, then bus-biassing can be used so that the quiescent bus value is loaded to represent an exception instruction format.

One of the major advantages of the detection mechanism placement is seen in the software development cycle. As a post-programming technique, this approach does not constrain the initial software, and the programmer need not be aware of its subsequent application. There is no pre-requisite programming requirement, and the original language of the software is immaterial because the utility is applied to the machine code. The method has wide application, existing software can be processed as a maintenance up-grade, or new software processed for immediate enhancement.

The use of the utility is proposed as part of an overall fault-tolerant strategy, an addition to and not a replacement for, other software and hardware techniques. Watchdogs alone would permit recovery after some interrupt interval, but the intervening period could consist of prolonged erroneous behaviour. The method proposed here reduces the mean period of erroneous behaviour and hence decreases the probability of catastrophic failure.

8. CONCLUSIONS

It has been shown that the injection of exception generating mechanisms into the machine code of a program can enhance the probability of recovery following a transient disturbance. This technique provides coverage for transient events which cause erroneous jumps into the program code. A particular program example shows performance improvement without the need for complex additional hardware. The technique is implemented by a software utility, PARUT, applied to existing program code. The method is therefore transparent to the programmer. The utility can be used to investigate other techniques of fault coverage, and can form part of an overall design strategy for reliable digital controllers.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the support of the UK Science and Engineering Research Council and the British Gas Engineering Research Station, Killingworth, Newcastle upon Tyne, England.

REFERENCES

[1] Buschke, H.A., *A Practical Approach to Testing Electronic Equipment for Susceptability to AC Line Transients.* IEEE Trans. Reliability, Vol. 37, No. 4, pp 355-359, 1988.

[2] Burton, P., *Designing Microprocessor-Based Equipment for Immunity from Electrical Interference.* Microprocessors and Microsystems, Vol. 12, No. 6, pp 309-316, 1988.

[3] Kotheimer, W.C., *The Source and Nature of Transient Surges.* IEEE Trans. Industrial Applications, Vol. 1A-13, No. 6, pp 501-503, 1977.

[4] Thurlow, M., *Susceptability Characterization of Microprocessor and LSI Technology.* Microprocessors and Microsystems, Vol. 12, No. 6, pp 317-322, 1988.

[5] Ball, M. & Hardie, F. *Effects and Detection of Intermittent Failures in Digital Systems.* AFIPS Conference Proceedings Fall Joint Computer Conference, Vol. 35, pp 329-335, 1969.

[6] McConnel, S.R. & Siewiorek, D.P. *C.vmp : The Implementation, Performance, and Reliability of a Fault Tolerant Multiprocessor.* Interim Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA 15213, USA, March 1978.

[7] Siewiork, D.P. & Swarz, R.S. *The Theory and Practice of Reliable System Design.* Digital Press, Bedford, MA, 1982.

[8] Iyer, R.K. & Rossetti, D.J. *A Statistical Dependancy of CPU Errors at SLAC.* Proc. FTCS-12, Santa Monica, CA, 1982.

[9] Lala, P.K. *Fault Tolerant and Fault Testable Hardware Design.* Prentice Hall International, New York, 1985

[10] Lu, D.J. *Watchdog Processors and Structural Integrity Checking.* IEEE Trans. Computers, C-31, No. 7, pp 681-685, 1982.

[11] Halse, R. *Fault Tolerance In Digital Controllers Using Software Techniques.* Ph.D. Thesis, University of Durham, England. 1984.

[12] Wingate, G.A.S. & Preece, C. *Transient Fault Recovery Assessment In 8 and 16 Bit Microprocessor Based Controllers In Embedded Systems.* Microprocessing and Microprogramming, Vol. 24, pp 775-782, 1988.

[13] Wingate, G.A.S. & Preece, C. *Enhanced Dependability for Microprocessor Based Controllers Susceptable to Transient Disturbances* , (to be published).

[14] Schutte, M.A. & Shen, J.P. *Processor Control Flow Monitoring Using Signatured Instruction Streams.* IEEE Trans. on Computing, C-36, No. 3, pp 264-276, 1987.

[15] Randell, B. *System Structure for Software Fault-Tolerance.* IEEE Software Engineering, Vol. 1, pp 220-232, 1975.

[16] Sosnowski, J. *Transient Fault Tolerance In Microprocessor Controllers.* IFIP/ IFAC Working Conference on Hardware and Software for Real Time Process Control (Warsaw) 1988, eds Zalewski, J. & Ehrenberger, W., North Holland Press, pp 189-195, 1989.

† *Special Note :* The probability of outcome for a particular state is represented by the vertical distance of the labelled outcome band shown on the graph.

FAULT TOLERANCE FOR MICROPROCESSOR-BASED CONTROLLERS SUSCEPTABLE TO TRANSIENT DISTURBANCES

G.A.S. Wingate & C. Preece

Abstract

This paper outlines the design of a microprocessor based controller with fault tolerance from transient disturbances. Such events can cause erroneous jumps within executing software. Fault tolerance is achieved through automatic enhancement of the application software. Performance issues are briefly discussed.

1. Introduction

Industrial controllers for monitoring and control are often based on microprocessors. The versatility offered to design aspects, both hardware and software, make them attractive in many industrial situations.

Operating conditions within industrial environments are often harsh. Transient disturbances such as mains power flucuations [1], and electro-magnetic radiation [2] may occur. Analogue systems effectively 'filter' these events without losing their control function by passing the transient event as a temporary signal discrepancy. Digital systems having a discrete nature are much more liable to lose all control function. Transients have been observed to cause between 80 and 90% of digital system failures [3,4]. It is important that digital systems should incorporate mechanisms for recovery from transient failures.

This paper considers those transient events which lead to corruption of bus information, memory contents, and register contents of a microprocessor system. Such corruption can induce erroneous behaviour whilst no permanent hardware damage occurs. If control of the microprocessor can be restored then overall system recovery is attainable.

Most techniques available replicate hardware and/or software, involve particular programming style, or require complex dedicated hardware. All these techniques are expensive in design and/or construction and are generally tailored to individual applications. The technique described here consists of automated software enhancement transparent to the programmer. The technique is directly applicable to a range of microprocessor systems and involves the self-detection of erroneous behaviour.

2. Erroneous Behaviour

We consider erroneous behaviour to be initiated by the corruption of the microprocessor's program counter. The erroneous behaviour produced by the microprocessor is characterised as a sequence of erroneous states terminated either by catastrophic failure or system recovery.

Execution states can be defined in terms of the outcome of each operation [5]. The possible states are defined below.

Erroneous execution may be within the used (program, data, or I/O reserved) area, or the unused area of the microprocessor address space. Recovery is attained through a restart state which vectors execution to a predefined memory location which holds the recovery routine.

School of Engineering and Applied Science,
University of Durham, DH1 3LE. England.

Non-Jump - leads to the program counter pointing to the location following a valid instruction.

Restart - leads to a jump to a predefined location in the address space.

Unspecified Jump - leads to a jump to a new location in the address space determined by local memory contents.

Return - leads to a jump to an address held in a stack.

Stop/Wait - leads to a cessation of processing ; and requires an interrupt or hardware reset to exit from this state.

3. Detection Technique

Erroneous execution can flow through both used and unused regions of the address space of a microprocessor system. Detection is based on the occurrence of a restart state during erroneous execution. Techniques for used and unused areas will now be presented.

3.1. Unused Area Detection

All execution within the unused area is defined to be erroneous and hence total detection coverage is required. Detection is achieved by ensuring all memory locations have the capability of generating restart state when accessed. Unused address space consisting of memory elements has every location filled with a restart outcome instruction [6]. Where there are no memory elements, a restart outcome is achieved through a simple hardware unit. This unit monitors the address bus and any illegal access results in the the unit developing an external reset for the microprocessor. The microprocessor will treat the external reset as a restart state.

3.2. Used Area Detection

Erroneous execution within the used area may flow through either program, data, or I/O reserved areas. Each will now be considered.

Program areas consist of opcodes and operands. Erroneous processing of an opcode will follow a valid execution path which is out of phase with the desired execution flow. Fault tolerance can be introduced by implementing software techniques such as recovery blocks or assertion tests [7].

Erroneous processing of an operand as an instruction will lead to erroneous execution dependent upon local memory contents. Such erroneous execution will follow paths, unpredictable to the programmer, through memory, leading to a danger of catastrophic failure. To detect this mode of erroneous behaviour, mechanisms are placed within memory so that any operand processed as a jump instruction will have at its destination a restart outcome instruction. This means that any operand being interpreted as an instruction and developing an erroneous jump will be followed by a restart and hence recovery. A design tool PARUT (Post-programming Automated Recovery UTility) automates this process [8].

Data and I/O reserved areas cannot have their structure altered in the same manner as the program areas. Erroneous execution within these areas may be detected through a watchdog timer.

4. Post-programming Automated Recovery UTility (PARUT)

A software tool called PARUT has been built to implement code enhancement. PARUT can provide automated code enhancement for a range of microprocessors. The tool works on machine code. There is no dependency on the original program language or pre-requisite programming style, hence the enhancement is transparent to the programmer. Enhancement may be provided

before software release or as a maintenance up-grade.

The performance improvement can be quantified with the application of PARUT. This is described in Reference 8. Many other fault tolerant techniques have been suggested but few offer analyses of their performance.

5. System Recovery

System recovery is attained through execution of the recovery routine accessed by the detection restart states. This routine may implement a number of recovery mechanisms including roll-back, roll-forward, or cold-start. The choice of recovery method will often be determined by the specific application of the microprocessor based controller.

6. Discussion

Detection is statistically very rapid. Early results show that detection within 500ns (10 instructions) has a likelihood of 99.9% [8]. A watchdog interval of 100ms, in a similar processor would permit approximately 2000 instructions to be erroneously processed before detection. Extended periods of erroneous execution increase the probability of catastrophic system malfunction.

Traditional methods of fault tolerance for this class of failure have involved large redundancy, or complex dedicated hardware, both very expensive and specific to a particular microprocessor. The technique presented here is easily transferable to other microprocessors, and its implementation, based on software, is automated.

7. Acknowledgements

The authors would like to thank the Science and Engineering Reasearch Council, and British Gas plc (Killingworth, Newcastle-upon-Tyne) for sponsoring this work.

8. References

[1] Buscke, H.A., *A Practical Approach to Testing Electronic Equipment for Susceptability to AC Line Transients.* IEEE Trans. Reliability, Vol. 37, No. 4, pp 355-359, 1988.

[2] Thurlow, M., *Susceptability Characterisation of Microprocessor and LSI Technology.* Microprocessors and Microsystems, Vol. 12, Noo. 6, pp 317-322, 1988.

[3] Siewiork, D.P. & Swarz, R.S., *The Theory and Practice of Reliable Systems Design.* Digital Press, Bedford, M.A., 1982.

[4] Iyer, R.K. & Rossetti, D.J., *Statistical Dependancy of CPU Errors at SLAC.* Proc. FTCS-12, Santa Monica, C.A., pp 363-372, 1982.

[5] Halse, R., *Fault Tolerance in Digital Controllers Using Software Techniques.* Ph.D. Thesis, University of Durham, England, 1984.

[6] Wingate, G.A.S. & Preece, C., *Transient Fault Recovery Assessment In 8 and 16 Bit Microrprocessor Based Controllers In Embedded Systems.* Microprocessing and Microprogramming, Vol 24, pp 775-782, 1988.

[7] Horning, J.J. et al, *A Program Structure for Error Detection an Recovery.* Lecture Notes in Computer Science, Vol. 16, (eds) Gelembe, E. & Kaiser, C., Springer-Verlag, pp 171-187, 1974.

[8] Wingate, G.A.S. & Preece, C., *Performance Evaluation of a New Design-Tool for Microprocessor Transient Fault Recovery.* Microprocessing and Microprogramming, Vol. 27, pp 801-808, 1989.

FAULT TOLERANCE FOR UNIPROCESSOR SYSTEMS

Wingate, G.A.S. & Preece, C.

School of Engineering and Applied Science
University of Durham

1. INTRODUCTION

Fault tolerance for system architectures is typically associated with multiple levels of redundancy, common examples are duplex, triplex, and quadruplex. Such system architectures are sometimes referred to as NMR (N-Modular Redundancy). Duplex systems can identify the module in error and switch to continue processing on the standby module. NMR systems, of a higher or than duplex, mask errant modules. Whilst these architectures offer very high reliability, their application also incurs in excess of 100% redundancy. The associated cost of this overhead may be significant and perhaps unacceptable for low budget systems. In such situations a uniprocessor fault tolerant approach may be appropriate.

2. FAULTS AND FAILURES

A significant hazard for all processor systems is that of temporary fault generation. Temporary faults, unlike permanent faults, incur no physical damage and have a limited duration and hence system recovery is possible without physical repair. Studies have suggested that temporary faults cause between 10 and 50 times more processor system failures than permanent faults.

Temporary faults can be classified as transient or intermittent. Transient faults occur unpredicably and are generated by environmental influences on the processor system such as electro-magnetic radiation, alpha-particles, power supply disturbances, and radio-frequency interference. Intermittent faults are recurring temporary faults and are indicative of imminent permanent fault generation.

3. CAPABILITY CHECKING

Namjoo & McCluskey [1982] first used the term 'capability checking' to describe a self-detection scheme that could be implemented by a uniprocessor system to identify errant processing. Since then, it has become evident in the literature that the collective application of a selection of capability checks provides the best method of achieving a highly reliable uniprocessor system.

The fault tolerant techniques proposed for capability checking in uniprocessor systems employ various strategies to detect erroneous processing. The strategies are based on the identification of different processing characteristics associated with erroneous behaviour. Implementation of these techniques incurs an overhead, physical (software and/or hardware) and/or processing (time). The various capability checks reported in the literature are summarised below..

Clive Preece is and Guy Wingate was formerly with the School of Engineering and Applied Science, University of Durham. Guy Wingate is now with ICI Engineering (Improved Manufacturing Systems Technology Group), Chilton House, Billingham, Cleveland.

A. Watchdog Timer

This is one of the most basic fault tolerant techniques, and involves a dedicated hardware unit called a 'watchdog timer'. The watchdog timer is incorporated into the processor system in such a way that failure of the processor to reset the watchdog timer periodically will result in the watchdog sending an alarm signal, representing indentified failure, back to the processor. Watchdog timers incur a small switching overhead. A disadvantage of the watchdog timer as a stand-alone technique is that the timer interval to detection can allow many hundreds of instructions to be processed haphazardly. The length and effect of this period of malfunction vary will vary between individual processor types and their applications.

B. Fetch Invalid Instruction

Most processor architectures have defined and undefined instruction opcodes. Some instruction sets, however, do not specify the action of their undefined instructions which may or may not have an operation. Two good examples are the Motorola 68000 in which all possible instruction opcodes have a specified operation, and the Intel 8085 which does not specify all its opcodes of which some undefined opcodes have useful operations [Denhardt, 1979]. In order to prevent the execution of an opcode of unknown operation, only defined instruction fetches should be allowed - all illegal instruction fetches should be detected. This technique requires additional decoder circuitry to be added to the uniprocessor system.

C. Invalid Opcode Address

Glaser & Masson [1982] proposed a SAFE ROM whereby an extra memory-bit is attached to each memory unit (usually a byte) to signify usage as an opcode of operand. Interpretation of an instruction activates decoding of the 'usage' bit and if the location is not specified as an opcode then erroneous processing is assumed to have been identified. The technique incurs a memory overhead and additional circuitry to decode the extra memory-bit. Furthermore, the techniques cannot be implemeneted in those locations in the address space implementing Random Access Memory (RAM) or non-existant memory.

D. Invalid Read/Write Within Permitted Memory

A dedicated hardware unit can be embedded within the processor system to ensures that a read is not made from a write only address, eg. a specified output port, or that a write is made to a read only address, eg. a Read Only Memory (ROM) location.

E. Unused Memory Access & Non-Existant Memory Access

A commonly used technique involves additional circuitry to check that the address bus does not carry signals accessing unused locations of physical memory or addresses without resident physical memory in the address space.

F. Invalid Branch

An invalid branch involves the incorrect interpretation of an instruction as a branch and should not be confused with interpreting a valid branch instruction whose destination is incorrect. A technique has been proposed by Wingate & Preece [1989] in which software detection blocks are strategically placed within the application code at locations identified as destinations of invalid branch instructions. A complementary hardware unit called an Access

Guardian detects unpermitted memory access in parts of the address space not implemeneted in physical memory.

G. Incorrect Sequence of Instructions

This method, commonly referred to a 'signature analysis', assigns tag values on a cyclic encoding of instruction sequences which are inserted within the software before implementation. Additional circuitry monitors the tags in the software, comparing the tag with a hardware generated tag. A favourable comparison verifies execution, whilst a mis-match of tag values signifies the identification of erroneous behaviour. A good review of this approach can be found in Mahmood & McCluskey [1988].

4. EFFECTIVENESS

The effectiveness of the fault tolerant techniques can be assessed using two parameters, fault coverage and fault latency. Fault coverage is derived from fault insertion experiments (injection, emulation, or simulation) expressing the percentage of faults detected. Fault latency describes the time interval that passes between the fault insertion and its detection.

4.1. Fault Coverage

Schmid et al [1982] identified erroneous program flow as the most prominent exposure feature of uniprocessor failure induced by a fault. They evaluated the individual performance of some of the techniques outlined above. The three most successful techniques, during fault simulations on the Z80 microprocessor, with 63%, 58%, and 56% fault coverage respectively were those based on detecting incorrect sequences of instructions, invalid opcodes, and unused memory access. Similar results are reported by Gunneflo et al [1989] and Li et al [1984] for the Z80 and SBR9000 processors respectively.

The reliability of a uniprocessor system can be improved by collectively applying several techniques. Table 1 collates data from three experiments evaluating different combinations of uniprocessor fault tolerant techniques. The fault coverage of the combinations is only an indication of the performance. There will be some variation in the results due to different methods of fault injection employed by the authors.

Source	Techniques Employed	Fault Coverage
Schmid et al [1982]	B, C, D, E, G	73%
Gunnelfo et al [1989]	C, D, E, G	79%
Madiera et al [1990]	A, B, D, E	75%

Table 1: Collective Application of Capability Checks

The overall fault coverage might appear low, but some account must be made for benign faults and those faults which generate data value errors and do not disrupt normal program action.

4.2. Fault Latency

Arlat et al [1990] describe a bimodal distribution of fault latencies, ie. there are two or more distinct classes of error manifestation. Chillarge & Bowen [1989] identify some faults to be dormant, such as stack corruption requiring particular processor activity to exercise the fault, whilst other faults induce 'fast failure'. It is therefore important that error detection be provided with a minimal fault latency in order to detect faults that would otherwise generate a 'fast failure'.

The fault tolerant techniques outlined above have a short fault latency, eg. the incorrect instruction sequence detection reported by Schmid et al [1982] ocurred with a mean latency of 8 μs. The precise latencies for each fault tolerant technique will vary depending on their host uniprocessor.

The collective application of a selection of fault tolerant techniques, whilst improving the fault coverage, incur a higher mean detection latency. Assessment of the detection latency is application specific, further details can be found in the references given in Table 1.

5. FUTURE WORK

Future work is needed to compare the individual and collective effectiveness of all the capability checks listed in this paper. This will facilitate the assessment of the benefits of strategically applying particular selections of capability checks to a uniprocessor application and the overheads they incur.

6. CONCLUSION

Processes or equipment that require low budget and yet reliable control can utilise fault tolerant uniprocessor systems. Such systems have high reliability without the order of magnitude redundancy associated with NMR architectures. Reliability can be further improved by implementing fault tolerant techniques, but this has the effect of increaseing fault latency. Nevertheless, fault latency is much less than that introduced by stand-alone watchdog timers traditionally associated with uniprocessor systems. In many industrial applications fault latency is not a critical factor because the equipment or process under control has a response time much longer than the fault latency.

REFERENCES

- Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E. & Powell, D., *Fault Injection for Dependability Validation: A Methodology and Some Applications.*, IEEE Trans. Soft. Engineering, Vol. 16, No. 2, 1990, pp 166-181.

- Chillarge, R. & Bowen, N.S., *Understanding Large System Failures – A Fault Injection Experiment.*, Int. Symp. on Fault Tolerant Computing, 1989, pp 94-99.

- Dehnhardt, W. & Sorensen, V.M., *Unspecified 8085 Op-Codes Enhance Programming.*, Electronics, 1979, pp 144-145.

- Glaser, R.E. & Masson, G.M., *The Containment Set Approach to Crash-Proof Microprocessor Controller Design.*, IEEE Trans. Computers, Vol. 31, No. 7, 1982, pp 689-692.

- Gunneflo, U., Karlson, J. & Torin, J., *Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation.*, Int. Symp. on Fault Tolerant Computing, 1989, pp 340-347.

- Li, K.W., Armstrong, J.R. & Tront, J.G., *An HDL Simulation of the Effects of a Single Event Upset on Microprocessor Program Flow.*, IEEE Trans. Nuclear Science, Vol. 31, No. 6, 1984, pp 1139-1144.

- Madiera, H., Quadros, G. & Silva, J.G., *Experimental Evaluation of a Set of Simple Error Detection Mechanisms.*, Microprocessing and Microprogramming, Vol. 30, 1990, pp 513-520.

- Namjoo, M. & McCluskey , E.J., *Watchdog Processors and Capability Checking.*, Proc. FTCS-12, 1982, pp 245-248.

- Namjoo, M. & McCluskey , E.J., *Concurrent Error Detection Using Watchdog Processors : A Survey.*, IEEE Trans. Computing, Vol. 37, No. 2, 1988, pp 160-174.

- Schmid, M.E., Trapp, R.L., Davidoff, A.E. & Masson, G.M., *Upset Exposure by Means of Abstarct Verification.*, Proc. FTCS-12, 1982, pp 237-244.

- Wingate, G.A.S. & Preece, C., *Performance Evaluation of a New Design-Tool for Microprocessor Transient Fault Recovery.*, Microprocessing and Microprogramming, Vol. 27, 1989, pp 801-808.

Microprocessing and Microprogramming 32 (1991) 861–868
North-Holland

ANALYSIS OF FAILURE DATA COLLECTED FROM A TMR MICROPROCESSOR CONTROLLER

Guy A.S. Wingate* and Clive Preece
School of Engineering and Applied Science
University of Durham, England

KEYWORDS: Microprocessor System Reliability, Failure Analysis, Real Time Systems, Temporary Faults.

Experimental failure data has been collected over two operational periods, each in excess of one year's duration, from a Triple Modular Redundancy (TMR) microprocessor controller based on the Intel 8085. Failures of embedded microprocessors are diagnosed as due to either temporary or permanent faults. There are few published studies covering temporary fault analysis of microprocessor failures; the research reported here is a valuable addition. Failures attributed to temporary faults are observed to occur approximately 40 times more frequently than those attributed to permanent faults. Further analysis of each data set reveals a very good correlation (0.992 and 0.995) to a constant failure rate, which is associated with an exponential inter-arrival distribution. This paper will be of interest to reliability engineers considering aspects of operational microprocessor system reliability.

1. INTRODUCTION

Microprocessor failures can be diagnosed as due to either permanent or temporary faults [9]. Permanent faults are physical defects, whilst temporary faults have a limited duration and do not incur physical damage. Temporary faults are often divided within the literature into transient and intermittent classes. Transient faults occur unpredictably and are attributed to temporary environmental conditions such as electrical power disturbances, electro-magnetic interference, radio-frequency interference, electro-static discharge, and alpha-particle radiation. Intermittent faults are recurring temporary faults and are associated with the imminent creation of a permanent fault (wear-out phenomena), or are faults whose activation is pattern sensitive.

Reliability engineers have observed electrical systems to exhibit a time dependent failure rate, referred to as the hazard rate, $Z(t)$. The Weibull function is widely used to describe the hazard rate as it varies during a systems lifetime (known as the 'Bathtub Curve'). The function is

$$Z(t) = \frac{\beta}{\alpha^{\beta}}.t^{\beta-1} \qquad (1.)$$

where α and β are known as the scale and shape parameters respectively.

The Bathtub Curve divides the lifetime of an electronic system into three phases. Firstly, the 'burn-in' phase involves the indentification of premature faults and is modelled by $\beta < 1$: hazard rate decreases. Secondly, the 'useful period' is marked by unpredictable faults, induced by component ageing and/or environmental stress, and is modelled by $\beta = 1$: constant hazard rate. A constant failure rate is a special case of the Weibull function implying an exponential distribution of inter-arrival failure times. Finally, the 'wear-out' phase occurs when faults due to component degrada-

* G.A.S. Wingate is now with ICI Engineering (Computer Aided Production), Chilton House, Billingham, Cleveland. U.K.

tion become increasingly significant compared with useful period faults, and is modelled by $\beta > 1$: hazard rate increases.

Temporary faults have a significant influence on microprocessor system reliability. A selection of monitored processor systems have shown that temporary faults are responsible for between 90 and 98% of system failures, see Table 1. In addition, temporary faults have been observed to occur as frequently as once every 100 hours during continuous operation. Microprocessor failure data collated by McConnel [6] and reported by Lin & Siewiorek [5] exhibits a Wiebull inter-arrival distribution with a decreasing hazard rate for failures attributed to temporary faults.

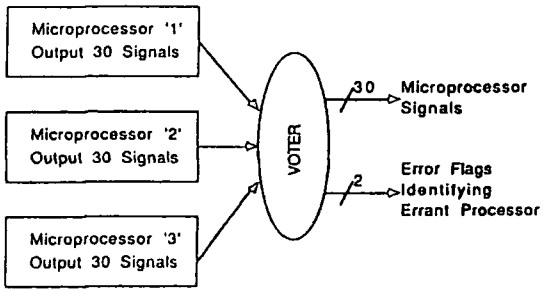


Figure 1. : TMR Controller

2. DESCRIPTION OF MICROPROCESSOR CONTROLLER

Failure data was collected from a TMR microprocessor system, based on the Intel 8085, used to control a gas governor system. The microprocessor system, see Figure 1., was designed and implemented by Pearson [8], and is briefly reviewed below.

The TMR architecture provides fault tolerance for embedded processor failures. A voter is implemented which compares thirty bus channel signals from each processor every 3 μs. The voter outputs the majority agreed value for each signal. Hence, the correct output is guaranteed when not more than one processor fails. The voter implemented by Pearson has additional circuitry which identifies single errant processors, the diagnosis being output on two error flag signals. When more than one processor fails concurrently, the majority decision process breaks down, and the voter outputs a signal to indicate voting failure. Some processor failures require multiple reset attempts (every $3\mu s$), however, if more than 100 reset attempts are required a steady state failure is assumed to have occured. The voter can operate at a maximum speed of 3 MHz and is the primary constraint on the microprocessor controller's operational speed, the Intel 8085 microprocessor being capable of operating at 8 MHz.

System	Source	Processor	Technology	Detection Mechanism	Hours Observed	Mean Time To Failure (Hours)		Temporary Faults (%)
						Temporary Faults	Permanent Faults	
CMUA	Fuller & Harbison [2]	PDP-10	ECL	Parity	5700	44	800-1600	94.8 - 97.3
Cm*	Siewiorek et al, [10]	LSI-11	NMOS	Diagnostics	15000	128	4200	97.0
C.vmp	Siewiorek et al, [11]	TMR LSI-11	NMOS	Crash	15000	97-328	4900	93.7 - 98.1
Telettra	Morganti et al, [7]	UDET 7116	TTL	Mismatch	N/A	80-170*	1300	88.4 - 94.2
SLAC	Iyer & Rossetti [3]	N/A	N/A	Diagnostics	26000	58	2300	97.5**
CMU-AFS	Lin & Siewiorek, [5]	MC 68010/20	NMOS	Diagnosis	212800***	201	6552	97.3

Notes: * Reported by McConnel [6]
** From which 85% were recovered
*** 13 applications monitored over 22 months

Table 1. : Observed Temporary & Permanent Faults

In addition, the microprocessor controller implements self-synchronising clock signals for the embedded microprocessors, a common reset signal, Random Access Memory (RAM) with code protection, Read Only Memory (ROM) duplicated - the reserve copy being selected by a watchdog timer, and an uninterruptable power supply.

Figure 2. : Example of Diagnosis Printout

3. DATA COLLECTION

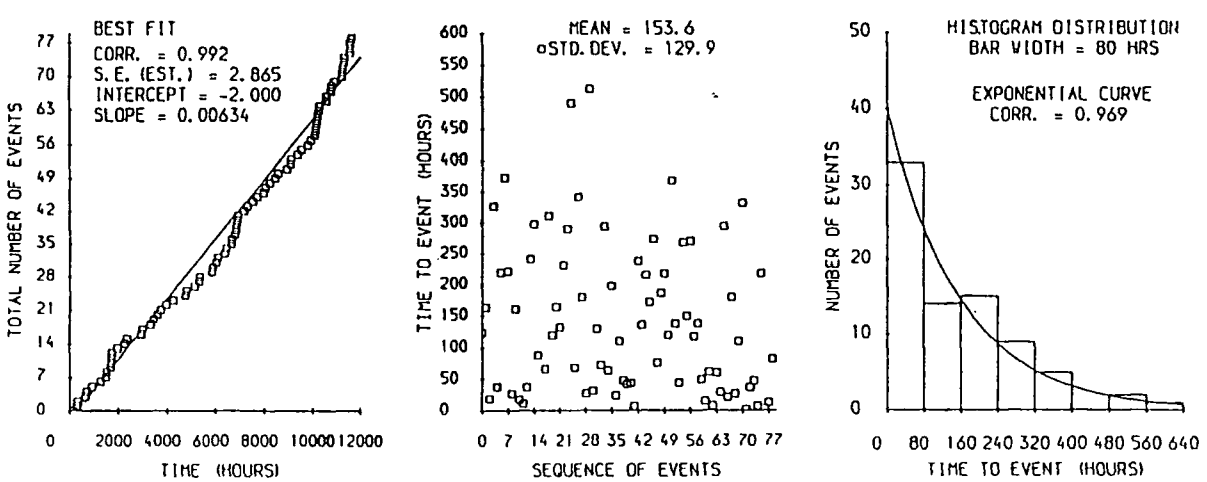
The microprocessor controller was in continuous operation from October 1983 to February 1985, and from June 1989 to July 1990. During these periods the controller retained operation through the automatic activation of fault tolerant mechanisms implemented by the system. Failure of the fault tolerant mechanisms resulted in a crash outcome. Instances of automatic recovery are assumed to restore a temporary fault condition because the failure incurred no physical damage. Crashes require operator intervention for recovery, via manual reset or repair, and hence because of their steady state failure condition are attributed to permanent faults.

The controller self-diagnoses failures from which system integrity can automatically be restored. Self-diagnosis information is output by the controller on an attached dedicated printer. A typical print-out is shown in Figure 2. The print-out shows the date and time of the incident, the identified errant processor, and the number of sequential reset attempts required to secure restored system operation.

The collected 1983/85 failure data was recorded over 12299 hours, noted 3 permanent faults and 79 temporary faults, and is referred to as data set 'A'. Similarly, the 1989/90 failure data is referred to as data set 'B' and recorded 3 permanent and 80 temporary faults inducing failure over 9360 hours. In both failure data sets, temporary faults are diagnosed as responsible for 96.3% of controller failures.

4. DATA ANALYSIS

The analysis of temporary fault failure sets 'A' and



(a) Linear Regression (b) Event Occurrence (c) Event Distribution

Figure 3. : Collected Failure Data Set 'A'

'B' are shown in Figures 3. and 4. respectively. Each figure has three graphs: (a) showing the cumulative number of events against operational time; (b) the failure time of each event as they occurred; and (c) the inter-arrival failure time distribution of the events.

4.1. Failure Occurrence

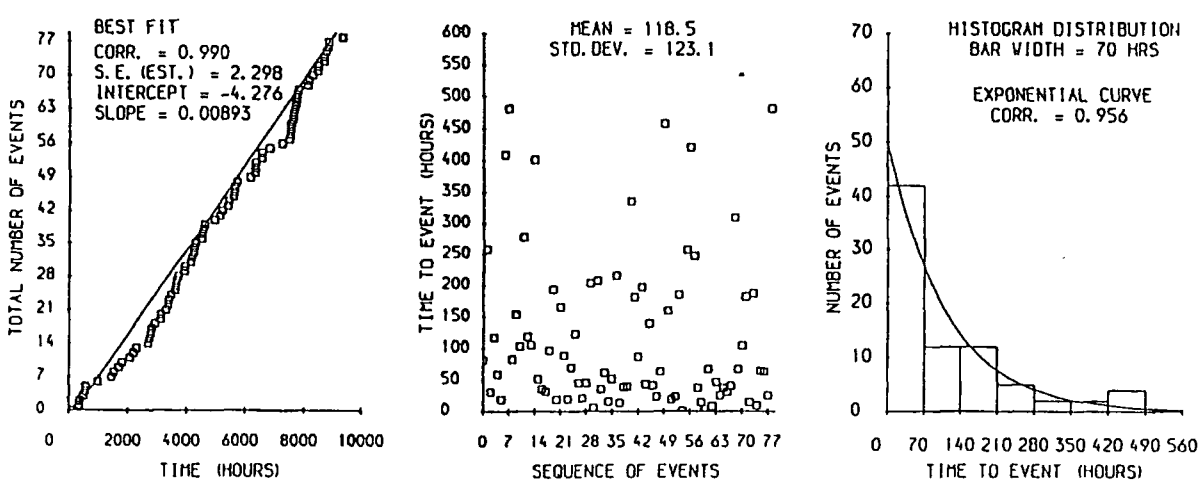
The observed failures caused by temporary faults are shown in Figures 3(a). and 4(a). for data sets 'A' and 'B' respectively. It is interesting to notice the step function in these figures, particularly of data set 'A'. The three steps of higher failure rate ocurrence in data set A are recorded for the months of December 1983, July 1984, and December 1984. The steps are harder to distinguish for data set 'B', but occur at the similar months of December 1989 and May 1990. Other processor systems have been observed to have a workload dependent failure rate [3, 4]. This, however, does not explain the observed failure rate step function for the TMR controller because its workload is deemed to be constant. The method of Lin and Siewiorek [5] applies Dispersion Frame Technique (DFT) to processor system failure data and observes an increased hazard rate associated with a intermittent fault. Such an observation does not fit the TMR controller data because the steps of increased failure rate do not terminate with a failure attributed to a permanent fault. An alternative explanation is that the failure rate is dependent on some external environmental influence but this suggestion cannot be substantiated.

Analysing linear regresssion on the whole of each failure data set still yields a very good correlation to a constant failure rate, λ_{LR}, despite the observed step function. Failure data sets 'A' and 'B' yield Mean Time To Failure (MTTF) parameters λ_{LR} of similar magnitude, 157.7 hours and 112.0 hours respectively. This compares favourably with the sample data MTTF of 147.8 hours and 117.6 hours for data sets A and B. Table 2 summarises the linear regression analysis on the whole of each failure data set.

4.2. MTTF Distribution

The inter-arrival times of failures diagnosed as due to temporary faults are shown in their sequential order of occurrence for failure data sets 'A' and 'B' in Figures 3(b). and 4(b). respectively. The plots suggest a memoryless failure mechanism because the scatter of inter-arrival failure times appears unchanged throughout the observed periods of controller operation.

A histogram distribution of the inter-arrival times for failures attributed to temporary faults for data sets 'A' and 'B' are shown in Figures 3(c). and 4(c). respectively. Histogram bars represent the number of failures



(a) Linear Regression (b) Event Occurrence (c) Event Distribution

Figure 4. : Collected Failure Data Set 'B'

that occur within sequential time intervals following the last failure event. The intervals, Δ, for the histogram, distribution plot are calculated using the following equation from Lewis [4]:

$$\Delta = r[1 + 3.3log_{10}(N_i)]^{-1} \quad (2.)$$

where Δ is considered to be a 'reasonable' interval, r is the range of values taken by the data, and N_i is the number of data items under analysis.

As, Δ is not usually a convenient value for plotting a histogram an approximate value Δ' is chosen. Table 3. summarises the Δ derivations and the choice of Δ'. The histogram interval for data set 'A' and 'B' is 80 and 70 hours respectively.

The distributions for data sets 'A' and 'B', yielding a correlation of 0.969 and 0.956 respectively with the Draper & Smith [1] non-linear regression method, are modelled by,

$$\gamma.exp\{-\lambda_{LR}.t\} \quad (3.)$$

where γ is given by,

$$\gamma = \lambda_{LR}.N_i.\Delta' \quad (4.)$$

The exponential distribution of the failure data validates the memoryless characteristic observation made earlier for the inter-arrival failure times.

4.3. Controller Unavailability

Some detected failures were only successfully recovered after multiple attempts (every 3 μs) to restore controller operation. Table 4 gives the down time distribution for the failures from which automatic recovery was achieved. Multiple attempts to restore the system integrity implies that the temporary fault causing the failure was still active when recovery action was initiated. Alternatively, a burst of temporary faults may have occurred. The nature of the collected failure data prevents further analysis or postulation.

Failure Data Set	r (hours)	N_i (hours)	Δ (hours)	Δ' (hours)
A	600	79	82.6	80
B	500	80	68.7	70

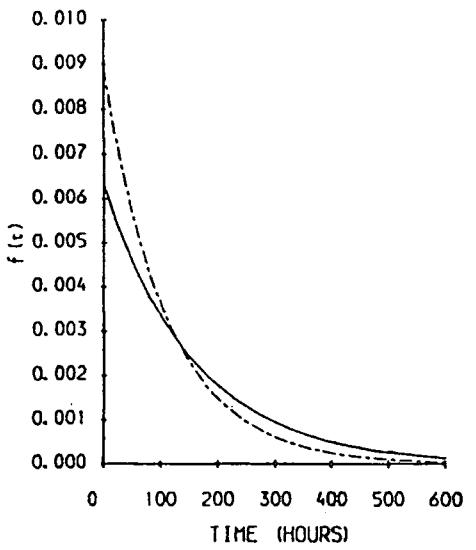
Table 3. : Histogram Distribution Interval

Down Time (μs)	Number of Failures	
	Data Set 'A'	Data Set 'B'
$0 \rightarrow 3$	59	61
$3 \rightarrow 6$	20	17
$6 \rightarrow 9$	0	1
$9 \rightarrow 12$	0	0
$12 \rightarrow 15$	0	0
$15 \rightarrow 18$	0	1
$18 \rightarrow \infty$	0	0

Table 4. : Failure Induced Unavailability

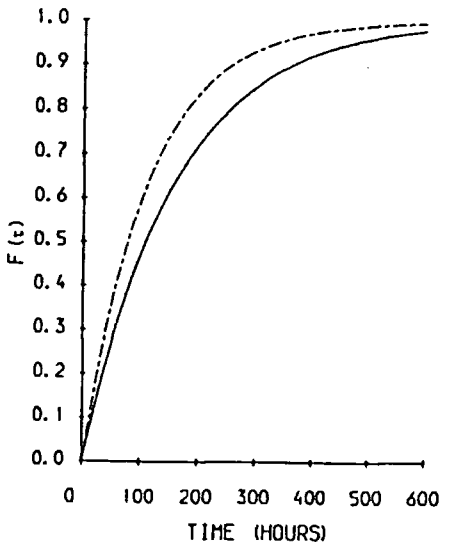
Failure Data Set	Linear Regression			Data Sample	
	λ_{LR} (hrs^{-1})	MTTF (hrs)	Correlation	λ_S (hrs^{-1})	MTTF (hrs)
A	0.00634	157.7	0.992	0.00676	147.8
B	0.00893	112.0	0.995	0.00850	117.6

Table 2. : Linear Regression Analysis



Solid line : from Data Set 'A'
Broken line : from Data Set 'B'

Figure 5 : Probability Density Function



Solid line : from Data Set 'A'
Broken line : from Data Set 'B'

Figure 6 : Cumulative Density Function

5. RELIABILITY ASSESSMENT

Siewiorek & Swarz [9] describe four evaluation parameters and their inter-relationship for failure distribution analysis: probability density function (pdf), cumulative density function (Cdf), reliability, and hazard function. Pdf, '$f(t)$', defines the probability of a failure occuring at a specific time. Cdf, '$F(t)$', defines the probability of failure occuring at or before a specific duration of operation. Reliability, '$R(t)$', is the probability of not observing a failure before a specific duration of operation. Finally, the hazard function, '$Z(t)$', is defined as the time dependent failure rate. Within this experiment the hazard function appears time independent, denoted by the constant λ, hence equation (1.) becomes,

$$Z(t) = \lambda \tag{5.}$$

Basic reliability theory gives the following parameter relationship,

$$F(t) = 1 - R(t) \tag{6.}$$

$$Z(t) = \frac{f(t)}{R(t)} \tag{7.}$$

The probability density function is given by converting the time intervals, used by the experimental results histogram, in equation (3.) into a general time parameter,

$$f(t) = \lambda_{LR}.exp\{-\lambda_{LR}.t\} \tag{8.}$$

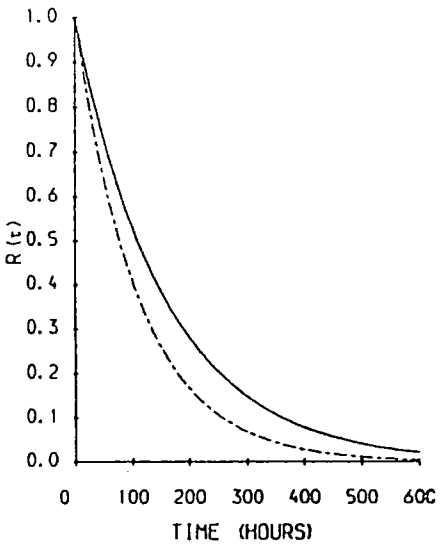
Reliability is evaluated using equation (7.),

$$R(t) = exp\{-\lambda_{LR}.t\} \tag{9.}$$

and the cumulative density function is evaluated from equation (6.),

$$F(t) = 1 - exp\{-\lambda_{LR}.t\} \tag{10.}$$

The performance parameters of pdf, Cdf, and reliability are evaluated for failures attributed to temporary faults and are plotted in Figures 5., 6., and 7. respectively. In each graph, data set 'A' evaluations are shown



Solid line : from Data Set 'A'
Broken line : from Data Set 'B'

Figure 7 : Reliability

as solid lines and data set 'B' as broken lines. The pdf plot presents the inter-arrival failure times modelled for the two data sets in Figure 3(c). and 4(c). Data set 'B' exhibits a higher failure rate than data set 'A' which may be due to the aging of the controller or varying working conditions. Figure 6. plots the Cdf for each data set, ie the running sum of the pdf coefficients, and clearly shows the higher failure rate observed for data set 'B'. The effect of the observed failure rates on the controller reliability for each data set is shown in Figure 7. Data set 'A' with the lower failure rate has a higher reliability.

6. DISCUSSION

The aim of the work was to observe the effects of temporary faults on a real control system. The results presented add to the limited body of published data in this area. The failure history of a TMR microprocessor controller has been collected over a 17 and a 13 month period. During each period approximately 80 failures occurred, of which 96% were diagnosed as due to temporary rather than permanent faults. The inter-arrival time of the failures, attributed to temporary faults, follows an exponential distribution. This observation suggests that the controller was operated during its useful period as specified by the Bathtub Curve commonly used by reliability engineers. These results demonstrate the validity of modelling the effects of temporary faults using techniques developed for permanent fault modelling, with constant hazard rates.

ACKNOWLEDGEMENTS

The authors wish to record their debt to former research students Dr. J.C. Pearson and Dr. R.G. Halse who helped to gather the data on which the results in this paper are based. Continued support from the Science and Engineering Research Council and from British Gas is also acknowledged.

REFERENCES

[1] Draper, N.R. & Smith, H., *Applied Linear Regression.*, Wiley and Sons, 1981.

[2] Fuller, S.H. & Harbison, S.P., *The C.mmp Multiprocessor.*, Technical Report CMU-CS-78-146, Carnegie-Mellon University, October 1978.

[3] Iyer, R.K. & Rossetti, D.J., *A Statistical Dependency of CPU Errors at SLAC.*, Proc. FTCS-12 (Santa Monica, CA), 1982, pp 363-372.

[4] Lewis, E.E., *Introduction to Reliability Engineering.*, John Wiley & Sons, New York, 1987.

[5] Lin, T-T. K. & Siewiorek, D.P., *Error Log Analysis : Statistical Modelling and Heuristic Trend Analysis.*, IEEE Trans. Reliability, Vol. 39, No. 4, 1990, pp 419-432.

[6] McConnel, S.R., *Analysis and Modelling of Transient Errors in Digital Computers.*, Ph.D. Disssertation, Carnegie-Mellon University, Pittsburgh, PA., 1981.

[7] Morganti, M., Coppadoro, G. & Ceru, S., *UDET 7116 - Common Control for PCM Telephone Exchange : Diagnostic Software Design and Availability Evaluation.*, Digest 8th Int. Conf. on Fault Tolerant Computing, 1978, pp 16-23.

[8] Pearson, J,C. *Reliability of Small Digital Controllers.*, Ph.D. Thesis, University of Durham, England, 1983.

[9] Siewiorek, D.P., & Swarz, R.S., *The Theory and Practice of Reliable Systems.*, Digital Pess, (Bedford, MA.), 1982.

[10] Siewiorek, D.P., Kini, V., Joobani, R., & Bellis, H., *A Case Study of C.mmp, Cm*, and C.vmp : Part 1 - Experiences with Fault Tolerance in Multiprocessor Systems.*, Proc. IEEE, Vol. 66, No. 10, 1978, pp 1178-1199.

[11] Siewiorek, D.P., Kini, V., Joobani, R., & Bellis, H., *A Case Study of C.mmp, Cm*, and C.vmp : Part 2 - Predicting and Calibrating Reliability of Microprocessor Systems.*, Proc. IEEE, Vol. 66, No. 10, 1978, pp 1200-1220.

[12] Woodbury, M.H. & Shin, K.G., *Measurement and Analysis of Workload Effects on Fault Latency in Real-Time Systems.*, IEEE Trans. Soft. Engineering, Vol. 16, No. 2, 1990, pp 212-216.

ERRATA:

Three typo-graphical errors appear in the above paper. Mean and Standard Deviation are incorrectly marked on Figure 4b & 5b: the values given there should be ignored. Table 3 should mark N_i as an integer. Correlation in Figure 4a should read '0.995'.